Imperative functional programs that explain their work

Imperative Functional Programs that Explain their Work

WILMER RICCIOTTI, University of Edinburgh
JAN STOLAREK, University of Edinburgh
ROLY PERERA, University of Edinburgh
JAMES CHENEY, University of Edinburgh

Program slicing provides explanations that illustrate how program outputs were produced from inputs. We build on an approach introduced in prior work by Perera et al. [2012], where dynamic slicing was defined for pure higher-order functional programs as a Galois connection between lattices of partial inputs and partial outputs. We extend this approach to *imperative functional programs* that combine higher-order programming with references and exceptions. We present proofs of correctness and optimality of our approach and a proof-of-concept implementation and experimental evaluation.

CCS Concepts: •**Software and its engineering** → **Semantics;** *Software testing and debugging;*

Additional Key Words and Phrases: program slicing; debugging; provenance; Galois connection

ACM Reference format:
Wilmer Ricciotti, Jan Stolarek, Roly Perera, and James Cheney. 2016. Imperative Functional Programs that Explain their Work. *PACM Progr. Lang.* 1, 1, Article 1 (January 2016), 29 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

When a program produces an unexpected result, experienced programmers often intuitively "run the program backwards" to identify the last part of the program that contributed to the output, to decide where to focus attention to find the bug. Effects such as mutable state (references, arrays) or exceptions can make this a nontrivial challenge. For example, in the following ML-like program (where `!y` means the value of reference cell `y`):

```
let f(x) = if (x == 0) then y := 6 * !z else (y := 84 / !z; w := g(!y + 12))
```

suppose we observe, on applying `f` to some argument, that afterwards `!y` has the value `42`, when we were expecting some other value. We might reasonably focus on the two possible assignments in the `else` branch, and hypothesise that perhaps the strange output resulted from `x` having the value `1` and reference cell `z` containing `2`. Of course, this reasoning relies on certain working assumptions, which may be invalid: we do not know whether `w` and `y` are aliases, we do not know whether `g` had side-effects, and we do not know the value of `x` that determined which branch was taken. Furthermore, if the above code executed inside an exception handler:

```
try f(1) with Division_by_zero -> y := 42
```

then there is another possible explanation: perhaps `!z` is `0`, so the attempt to divide `84` by zero failed, raising an exception whose handler eventually assigned `42` to `y`. Alternatively, such an exception could have been raised from within the function `g`. This illustrates that the exact sequence of

DOI: 10.1145/nnnnnnn.nnnnnnn

events leading to an unexpected result may be impossible to determine based solely on the output. Often, programmers resort to interactive debugging, or even manually adding `print` statements, to observe the actual control flow path.

Backwards reasoning to identify the parts of a program that may have contributed to the output is the basis of a popular analysis technique called *program slicing*, invented by Weiser [1981]. In general, slicing techniques take a program and a *slicing criterion* describing the part of the program's output or behaviour of interest, and produce a *slice*, or subset of the program text, that identifies the parts of the program relevant to the criterion, omitting (as much as possible) parts that were not relevant. A typical slicing criterion might consist of a source location P (expression or statement) and a set of variables, and the slice would contain those parts of the program deemed to be relevant to the values of those variables at P.

Slicing techniques can be divided into two broad categories: *static* slicing conservatively analyses all possible executions of the program and identifies those parts which *potentially* influence the slicing criterion, whereas *dynamic* slicing identifies those parts of the program which *do* influence the slicing criterion on a particular execution. Because dynamic slicing analyses a specific run, it is especially useful for debugging and testing, where the goal is to understand a particular scenario or test case as precisely as possible.

In the above example, a static slicing algorithm might deem it unsafe to exclude any part of the program from the static slice because any part of the program could have affected the final value of `y`. Using dynamic slicing, we can increase precision by considering exactly what happened, given the actual store that the program ran in. For example, if we shade elided parts of the program, one possible slice with respect to output observation `!y = 42` is:

```
let f(x) = if (x == 0) then y := 6 * !z else (y := 84 / !z; w := g(!y + 12))
try f(1) with Division_by_zero -> y := 42
```

This slice shows that the `Division_by_zero` exception was raised and handled (so `!z` must have been `0`). On the other hand, if slicing yields:

```
let f(x) = if (x == 0) then y := 6 * !z else (y := 84 / !z; w := g(!y + 12))
try f(1) with Division_by_zero -> y := 42
```

then this means no exception was raised (so `!z` was not `0`), `w` did not alias `y`, and the final assignment of `42` to `y` must have been a side-effect of `g` which used its prior value of `84 / !z`.

The slicing used in these examples is *backward* slicing, which works back from outputs to contributing program parts. Bergeretti and Carré [1985] introduced the complementary technique of *forward* slicing, which works forward from program parts to outputs they contribute to. Forward slicing corresponds to the kind of informal reasoning programmers use during debugging when they try to understand the *consequences* of a fragment of code. Needless to say, program slicing, both forward and backward, has turned out to have many applications in program transformation and optimisation besides debugging, and has been researched very thoroughly in the context of mainstream imperative and object-oriented programming languages such as C/C++ and Java; Xu et al. [2005] cite over 500 papers on slicing. Slicing for functional programs, however, has received comparatively little attention. Biswas [1997] developed slicing techniques for a higher-order ML-like language, including references and exceptions, but only with respect to the whole program result as slicing criterion. Other authors have investigated slicing for pure or lazy languages such as Haskell or Curry [Ochoa et al. 2008; Rodrigues and Barbosa 2007; Silva and Chitil 2006].

Perera et al. [2012] introduced a new approach to dynamic slicing for (pure) functional programs where the slicing criteria take the form of *partial values*, allowing for fine-grained slicing particular to specific sub-values. In this approach, input and output values may be partly elided in the same

way as program slices, with *holes* (written □) intuitively corresponding to parts of programs or values that are not of interest. Perera et al. showed how to extend the usual semantics of pure, call-by-value programs with rules for □ to construct a *Galois connection* for program slicing. The forward component of the Galois connection maps a partial input x to the *greatest* partial output y that can be computed from x; the backward component of the Galois connection maps a partial output y to the *least* partial input x from which we can compute y. (Note that this use of Galois connections for *dynamic* slicing is unrelated to their widespread application to *static* analysis techniques such as abstract interpretation [Cousot and Cousot 1977; Darais and Van Horn 2016] or gradual typing [Garcia et al. 2016].) Once the forward slicing semantics is defined, the behaviour of backward slicing is uniquely determined as the *lower adjoint* of the forward semantics, whose existence is established using standard lattice-theoretic techniques. Perera et al. also showed how to compute such slices efficiently, through a semantics instrumented with *traces* that record details of program execution.

In this paper, we build on this fine-grained approach and address the challenges of adapting it to *imperative functional programming*: that is, programming with higher-order functions, references, and exceptions. We focus on a simplified, ML-like core language, so our approach is immediately relevant to languages such as Standard ML, OCaml, Scheme, or F#. Types do not feature in our approach, so our results should apply to both statically typed and dynamically typed languages. However, this paper focuses on foundational aspects of slicing for imperative functional programming, via a core language and proof of correctness, and more work would need to be done to develop a full-scale slicing tool for a mainstream language.

To illustrate how our approach compares to previous work, here is a (contrived) program that our slicing system (and no prior work) handles, and its slice explaining why an exception was raised:

```
let a = ref 1 in
let b = ref 2 in
map (fun c -> b := !b - 1 ; 1/!c)
    [a,b]
```

```
let a = ref 1 in
let b = ref 2 in
map (fun c -> b := !b - 1 ; 1/!c)
    [a,b]
```

This program does not return normally; it raises an exception because of the attempt to divide 1 by zero. Our approach produces a backward slice (shown on the right) as an explanation of the exception. It shows that the exception was raised because of the attempt to divide 1 by `!c`, when `!c` was zero after `b` was decremented the second time. In Biswas' approach (the only prior work to handle higher-order functions, references, and exceptions), the whole program would have to be included in the slice: without the ability to represent slicing criteria as partial values, there is no way to capture the partial usage of the list value supplied to `map` (in other words, the fact that only part of the list was needed to produce the exception). Our approach, in contrast, allows us to slice each sub-computation with respect to a precise criterion reflecting exactly the contribution required for that step. Here, it is safe to slice away the expression that defined `a` as long as we remember that it did not throw an exception. The main contribution of this paper is showing how to make the above intuitions precise and extend the Galois connection approach to higher-order programming with effects.

1.1 Contributions and Outline

In the rest of this paper, we present the technical details of our approach together with a proof-of-concept implementation. In detail, our contributions are as follows:

- (Section 2) We first review (what we call) *Galois slicing*, the Galois connection approach to dynamic slicing introduced by Perera et al. [2012], illustrated using a simple expression

language. In particular, we offer a direct argument showing that any pair of forward and backward slicing functions that satisfy appropriate optimality properties form a Galois connection (Prop. 2.1).
- (Section 3) We extend Perera et al.'s core language TML (Transparent ML) with exceptions and references, and call the result iTML ("imperative TML"). Our core language differentiates between pure *expressions* and *computations* that may have side effects. We define partial values, expressions, and traces, and state the rules (similar to those of Perera et al.) for slicing pure expressions.
- (Section 4) We define forward and backward slicing semantics for effectful computations, which abstract away information about results of sub-computations that return normally and do not affect the slicing criterion. The Galois connection for a computation includes the proof term, or *trace*, that explains how the result was computed, allowing the slicing semantics to compute a *least explanation* optimised to a particular partial outcome. Thm. 4.6 proves this and is our main technical contribution.
- (Section 5) We consider several natural extensions: slicing in the presence of mutable arrays, while-loops and sequential composition, and give illustrative examples.
- (Section 6) We present a proof-of-concept implementation of our approach in Haskell, discuss a more substantial example, and make preliminary observations about performance.

In the remainder of the paper, we discuss related work and future directions in greater detail, and summarise our findings. Detailed proofs of our main results, some straightforward rules, and an extended example are all included in the appendices of the full version of the paper.

2 BACKGROUND: GALOIS SLICING

In this section we recapitulate the Galois connection approach to dynamic slicing introduced for pure functional programs by Perera et al. [2012], using a simple expression language as an example. We call their approach *Galois slicing*. We then discuss the challenges to adapting this framework to references and exceptions; the rest of the paper is a concrete instantiation of this framework in that setting.

2.1 Ordered sets, lattices and Galois connections

We first review ordered sets, lattices and Galois connections. An *ordered set* $(P, \leq)$ is a set P equipped with a partial order $\leq$, that is, a relation which is reflexive, transitive and antisymmetric. A function $f : P \to Q$ between ordered sets $(P, \leq_P)$ and $(Q, \leq_Q)$ is *monotone* if it preserves the partial order, i.e. if $x \leq_P x'$ implies $f(x) \leq_Q f(x')$ for all $x, x' \in P$. The *greatest lower bound* (or *meet*) of two elements $x, y \in P$ (if it exists) is written $x \sqcap y$ and is the largest element of P such that $x \geq x \sqcap y \leq y$. The *least upper bound* (or *join*) $x \sqcup y$ is defined dually as the least element satisfying $x \leq x \sqcup y \geq y$ when it exists. Likewise we write $\bigsqcup S$ or $\bigsqcap S$ for the least upper bound or greatest lower bound of a subset S of P, when it exists.

A *lattice* is an ordered set in which all pairwise meets and joins exist. A lattice is *complete* if all subsets have a meet and join, and *bounded* if it has a least element $\bot$ and a greatest element $\top$. All finite lattices are complete and bounded, with $\bot = \bigsqcap P$ and greatest element $\top = \bigsqcup P$. A function $f : P \to Q$ where Q is a lattice is *finitely supported* if $\{x \in P \mid f(x) \neq \bot\}$ is finite, where $\bot$ is the least element of Q. Given $x \in P$, we define the *lower set* $\downarrow(x) = \{x' \in P \mid x' \leq x\}$ of all elements below x. We introduce the notion of a *partonomy* for a set X, which we define to be a partial order $P_X \supseteq X$ such that every element of X is maximal in P_X and $\downarrow(x)$ is a finite lattice for all elements $x \in X$.

Given ordered sets $(P, \leq_P)$ and $(Q, \leq_Q)$, a *Galois connection* is a pair of (necessarily monotone) functions $(f : P \to Q, g : Q \to P)$ that satisfy

$$f(p) \leq_Q q \iff p \leq_P g(q)$$

The function f is sometimes called the *lower adjoint* and g the *upper adjoint*. We say f and g are *adjoint* (written $f \dashv g$) when (f, g) is a Galois connection with lower adjoint f and upper adjoint g.

2.2 Galois connections for slicing

Now we show how to interpret *partial programs* and *partial values* in this setting of lattices and Galois connections. We consider a simple language of expressions with numbers, addition, and pairs.

$$e \quad ::= \quad n \mid e_1 + e_2 \mid (e_1, e_2) \mid \texttt{fst}\ e \mid \texttt{snd}\ e \qquad v \quad ::= \quad n \mid (v_1, v_2)$$

Suppose a standard big-step semantics is defined using a judgment $e \Rightarrow v$, as follows:

$$\frac{}{n \Rightarrow n} \qquad \frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{e_1 + e_2 \Rightarrow n_1 +_{\mathbb{N}} n_2} \qquad \frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{(e_1, e_2) \Rightarrow (v_1, v_2)} \qquad \frac{e \Rightarrow (v_1, v_2)}{\texttt{fst}\ e \Rightarrow v_1} \qquad \frac{e \Rightarrow (v_1, v_2)}{\texttt{snd}\ e \Rightarrow v_2}$$

A *partial* expression is an expression that may contain a *hole* $\square$. We define an order on partial expressions as follows. (From now on we write the partial orders simply as $\sqsubseteq$, omitting the subscripts.)

$$\frac{}{\square \sqsubseteq e} \qquad \frac{}{n \sqsubseteq n} \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{e_1 + e_2 \sqsubseteq e_1' + e_2'} \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{(e_1, e_2) \sqsubseteq (e_1', e_2')} \qquad \frac{e \sqsubseteq e'}{\texttt{fst}\ e \sqsubseteq \texttt{fst}\ e'}$$

$$\frac{e \sqsubseteq e'}{\texttt{snd}\ e \sqsubseteq \texttt{snd}\ e'}$$

This relation is simply the compatible partial order generated by $\square \sqsubseteq e$, and thus is reflexive, transitive and antisymmetric. It is easy to verify that meets $e_1 \sqcap e_2$ exist for any two partial expressions; for example, $(1, 2) \sqcap (1, 2 + 2) = (1, \square)$; however, joins do not always exist, since for example there is no expression e satisfying $(1, 2) \sqsubseteq e \sqsupseteq (1, 2 + 2)$. Nevertheless, if we restrict attention to the (finite) set $\downarrow(e)$ of prefixes of a given expression e, joins do exist; that is, $\downarrow(e)$ is a (finite) lattice, with $\bot = \square$ and $\top = e$. For example, $\downarrow((1, 2)) = \{\square, (1, \square), (\square, 2), (1, 2)\}$, and the join $(1, \square) \sqcup (\square, 2)$ is defined and equal to $(1, 2)$. Since all values happen to be expressions, we can also derive lattices $\downarrow(v)$ of prefixes of a given value v, obtaining $\sqsubseteq$ by restriction from the partial order on expressions.

Suppose we have some "computation" relation $C \subseteq X \times Y$, which for now we assume to be deterministic. Given a particular computation $(x, y) \in C$, we would like to define a technique for slicing that computation. We start by defining partonomies P_X and P_Y for X and Y. Given a partial input $x' \in \downarrow(x)$, it is natural to expect there to be a corresponding partial output $y' \in \downarrow(y)$ that shows how much of the output y can be computed from the information available in x'. Suppose such a function $\mathsf{fwd} : \downarrow(x) \to \downarrow(y)$ is given. We already know that given *all* of the input ($x \in \downarrow(x)$) we can compute all of the output y. Thus if fwd computes as much as possible, then $\mathsf{fwd}(x)$ should certainly be y. By the same token, it seems reasonable that given *none* of the input ($\square \in \downarrow(y)$) we should be unable to compute any of the output, so that $\mathsf{fwd}(\square) = \square$. More generally, we would like to be able to compute *partial* output from *partial* input, so that for example $\mathsf{fwd}(\square, 2 + 2) = (\square, 4)$ since we can compute the second component of a pair without any knowledge of the first. Finally, a reasonable intuition seems to be that fwd should be monotone, since learning more information about the input should not make the output any less certain.

As a concrete example of such a fwd function, we can extend the (deterministic) evaluation relation $\Rightarrow$ defined earlier for expressions to *partial* expressions as follows:

$$\frac{}{n \nearrow n} \qquad \frac{e_1 \nearrow n_1 \qquad e_2 \nearrow n_2}{e_1 + e_2 \nearrow n_1 + n_2} \qquad \frac{e_1 \nearrow v_1 \qquad e_2 \nearrow v_2}{(e_1, e_2) \nearrow (v_1, v_2)} \qquad \frac{e \nearrow (v_1, v_2)}{\texttt{fst}\ e \nearrow v_1} \qquad \frac{e \nearrow (v_1, v_2)}{\texttt{snd}\ e \nearrow v_2}$$

$$\frac{}{\square \nearrow \square} \qquad \frac{e_1 \nearrow \square}{e_1 + e_2 \nearrow \square} \qquad \frac{e_1 \nearrow v_1 \qquad e_2 \nearrow \square}{e_1 + e_2 \nearrow \square} \qquad \frac{e \nearrow \square}{\texttt{fst}\ e \nearrow \square} \qquad \frac{e \nearrow \square}{\texttt{snd}\ e \nearrow \square}$$

If the parts needed to perform a given evaluation step are present, we behave according to the corresponding $\Rightarrow$ rule (top row). Otherwise we use a rule from the bottom row: if the expression itself is missing ($\square$), or if a needed intermediate result (e.g. the tuple value for a projection, or an argument to an addition) is $\square$, then the result is again $\square$. So, for example, $(\square + 1, 1) \nearrow (\square, 1)$. For a given $e \Rightarrow v$ and $e' \in \downarrow(e)$, it is not possible for $e' \nearrow$ to get stuck, for example by trying to add an integer to a pair, and indeed one can verify that these rules define a total, monotone function $\mathrm{fwd}_e : \downarrow(e) \to \downarrow(v)$ such that $\mathrm{fwd}_e(e') = v' \iff e' \nearrow v'$. The function fwd_e computes the *forward slice* of $e' \in \downarrow(e)$, namely that portion of v which can be computed using only the information in e'.

Now, given $\mathrm{fwd}_e : \downarrow(x) \to \downarrow(y)$, we would like to define a converse mapping $\mathrm{bwd}_e : \downarrow(y) \to \downarrow(x)$ which computes the *backward slice* for $y' \in \downarrow(y)$, namely a partial input large enough to compute y'. That is, we want bwd_e to satisfy $y' \sqsubseteq \mathrm{fwd}_e(\mathrm{bwd}_e(y'))$ for any $y' \in \downarrow(y)$. We call this property *consistency*.

There may be many consistent choices of bwd_e. As an extreme example, the constant function $\mathrm{bwd}_0(y') = x$, which simply produces the full input for any output prefix, is consistent. However, as a slicing function it is singularly useless: it treats all parts of the input as relevant, failing to take advantage of the fact that not all of the output was required. Ideally, therefore, we would like bwd to satisfy the following *minimality* property:

$$\mathrm{bwd}_e(y') = \bigsqcap \{x' \mid y' \sqsubseteq \mathrm{fwd}_e(x')\} \tag{1}$$

This (together with consistency) says that $\mathrm{bwd}_e(y')$ is the smallest part of the input that provides enough information to recompute y' using fwd_e.

Now, if bwd_e is the lower adjoint of fwd_e (if bwd_e and fwd_e form a Galois connection $\mathrm{bwd}_e \dashv \mathrm{fwd}_e$) then the monotonicity, consistency and minimality properties follow by standard arguments [Davey and Priestley 2002]. More surprisingly, these properties suffice to ensure that $\mathrm{bwd}_e \dashv \mathrm{fwd}_e$:

Proposition 2.1. *Given complete lattices P, Q, suppose $g : Q \to P$ is monotone and $f : P \to Q$ is consistent and minimal with respect to g. Then they form a Galois connection $f \dashv g$.*

Proof. First suppose $f(p) \sqsubseteq q$. Then $p \sqsubseteq g(f(p)) \sqsubseteq g(q)$ by consistency of f and monotonicity of g. This proves that $f(p) \sqsubseteq q \Rightarrow p \sqsubseteq g(q)$. For the converse, assume that $p \sqsubseteq g(q)$. Then

$$f(p) = \bigsqcap \{q' \mid p \sqsubseteq g(q')\} \sqsubseteq q$$

where the equality is the minimality of f and the inequality holds because $p \sqsubseteq g(q)$. This proves that $p \sqsubseteq g(q) \Rightarrow f(p) \sqsubseteq q$, so $f \dashv g$. □

It may appear difficult to design an adjoint pair of functions fwd_e and bwd_e, or even to be sure that one exists for a given candidate definition of fwd_e. Luckily, another standard result applies: if P is a complete lattice then $g : Q \to P$ has a lower adjoint $f : P \to Q$ if and only if g preserves meets, that is, $g(\bigsqcap S) = \bigsqcap \{g(s) \mid s \in S\}$. For finite lattices, it suffices to consider only binary meets and the

top element: $g(q_1 \sqcap q_2) = g(q_1) \sqcap g(q_2)$ and $g(\top_Q) = \top_P$. Moreover, the lower adjoint $f : P \to Q$ is uniquely determined by the minimality equation. (Dually, any *join*-preserving function between complete lattices uniquely determines a "maximising" meet-preserving function as its upper adjoint, but for an evaluation relation it seems more natural to start with forward slicing and induce the backward-slicing function.)

Of course, for a given computation there may be more than one choice of lattice structure for the input and output, and there may also be more than one natural choice of meet-preserving forward slicing. Once such choices are made, however, a minimising backward-slicing function is determined by the forward semantics. Nevertheless, there are two considerations (beyond meet-preservation) that make defining a suitable forward-function non-trivial: first, the availability of an efficient technique for computing the backward-slicing lower adjoint, and second, the precision of the forward-slicing function (which in turn determines the precision of backward slicing).

To see the first point, we return to our simple expression language. It is easily verified that fwd_e is indeed meet-preserving in that $\mathsf{fwd}_e(e_1 \sqcap e_2) = \mathsf{fwd}_e(e_1) \sqcap \mathsf{fwd}_e(e_2)$ and $\mathsf{fwd}_e(e) = v$. Since $\downarrow(e)$ is a finite lattice, it follows that fwd_e has a lower adjoint $\mathsf{bwd}_e : \downarrow(v) \to \downarrow(e)$, which satisfies the minimality property (Equation 1). The minimality property alone, however, is not suggestive of an efficient procedure for computing bwd_e. Read naively, it suggests evaluating fwd_e on all partial inputs (of which there may be exponentially many in the size of e), and then computing the meet of all partial inputs e' satisfying $v' \sqsubseteq \mathsf{fwd}(e')$.

Perera et al. showed how the lower adjoint can be efficiently computed by adopting an algorithmic style "dual" to forward slicing: whereas forward slicing *pushes* □ *forward* through the computation, erasing any outputs that depend on erased inputs, backward slicing can be implemented by *pulling* □ *back* through the computation, erasing any parts of the input which are not needed to compute the required part of the output. In their functional setting, this involves using a *trace* of the computation to allow the slicing to proceed backwards. In the toy language we consider here, the expression itself contains enough information to implement backward slicing in this style. The following definition illustrates:

$$\begin{array}{rclrcl} \mathsf{bwd}_n(n) & = & n & \mathsf{bwd}_{e_1+e_2}(n) & = & \mathsf{bwd}_{e_1}(\mathsf{fwd}(e_1)) + \mathsf{bwd}_{e_2}(\mathsf{fwd}(e_2)) \\ \mathsf{bwd}_{\mathtt{fst}\ e}(v) & = & \mathsf{bwd}_e(v, \square) & \mathsf{bwd}_{(e_1,e_2)}(v_1, v_2) & = & (\mathsf{bwd}_{e_1}(v_1), \mathsf{bwd}_{e_2}(v_2)) \\ \mathsf{bwd}_{\mathtt{snd}\ e}(v) & = & \mathsf{bwd}_e(\square, v) & \mathsf{bwd}_e(\square) & = & \square \end{array}$$

The interesting cases are those for addition and projections. For addition, we continue slicing backwards, using the original values of the subexpressions. This still leaves something to be desired, since we use fwd to reevaluate subexpressions e_1 and e_2. (It is possible to avoid this recomputation in the trace-based approach by recording extra information about the forward evaluation that bwd can use.) For projections such as `fst` e, we slice the subexpression e with respect to partial value $(v, \square)$, expressing the fact that we do not need the second component. If the partial output is a hole, then we do not need any of the input to recompute the output. For example, suppose $e = (1, \mathtt{fst}\ (1, 2) + 3)$. Then $\mathsf{bwd}_e(\square, 4)$ yields $(\square, \mathtt{fst}\ (1, \square) + 3)$ because we do not need the first 1 or the 2 to recompute the result. One of the key challenges we address in this paper is adapting this algorithmic style to deal with imperative features like exceptions and stores.

To see the importance of precision for forward slicing, consider the following alternative slicing rules for pairs:

$$\frac{e_1 \nearrow \square}{(e_1, e_2) \nearrow (\square, \square)} \qquad \frac{e_1 \nearrow v_1 \qquad e_2 \nearrow v_2}{(e_1, e_2) \nearrow (v_1, v_2)}\ v_1 \neq \square$$

Evaluation goes left-to-right, and so naively we might suppose that if we know nothing about the first component, we should not proceed with the second component. The forward-slicing function

for a given computation still preserves meets, and thus has a backward-slicing lower adjoint. However, again supposing $e = (1, \texttt{fst}\ (1, 2) + 3)$, we have $\mathrm{fwd}_e(\square, \texttt{fst}\ (1, 2) + 3) = (\square, \square)$ by the first rule above. If we then use this partial output to backward slice, we find $\mathrm{bwd}_e(\square, \square) = (\square, \square)$: if all we need of the output is the fact that it is a pair, then all we actually needed of the program was the fact that it computes a pair. Thus the "round trip" $\mathrm{bwd}_e(\mathrm{fwd}_e(e'))$, technically a *kernel operator*, reveals *all* the parts of e that are rendered irrelevant as a consequence of retaining only the information in e', and here $\mathrm{bwd}_e(\mathrm{fwd}_e(e'))$ reveals a (spurious) dependency of the second component of the pair on the first. This motivates the more precise pair-slicing rule we first presented, which sliced each component independently. A key design criterion for forward-slicing therefore is that it only reflect genuine dependencies, capturing specifically how input was consumed in order to produce output. Defining suitably precise forward slicing in the presence of stores and exceptions is another of the key challenges we address in this paper.

2.3 Summary

To summarise, the Galois slicing framework involves the following steps:

- Given sets of expressions, values and other syntactic objects, define partonomies such that the set of prefixes of each object forms a finite lattice.
- Given a reference semantics for the language, say a deterministic evaluation relation $\Rightarrow \subseteq X \times Y$, define a family of meet-preserving functions $\mathrm{fwd}_x : \downarrow(x) \to \downarrow(y)$ for every $x \Rightarrow y$. By the reasoning given above, fwd_x has a lower adjoint $\mathrm{bwd}_x : \downarrow(y) \to \downarrow(x)$ that computes the least slice of the input that suffices to recompute a given partial output.
- Define a procedure for computing a backward slice, typically by running back along a trace of $x \Rightarrow y$, and show that the procedure computes the lower adjoint bwd_x.

The framework describes a design space for optimal slicing techniques for a given language: we have latitude to decide on suitable lattices of partial inputs and outputs and a suitable definition of forward slicing, as long as it is compatible with ordinary evaluation and is a meet-preserving function. The definition of forward slicing must be precise enough to reflect accurately how information in the input is consumed during execution to produce output, and there may be different notions of slicing suitable for different purposes. Once these design choices are made, the extensional behaviour of an optimal backward slicing bwd is determined, and the remaining challenge is to find an efficient method for computing backward slices, using traces where appropriate.

3 CORE CALCULUS AND COMMON CONCEPTS

We now introduce the core calculus iTML, "Imperative Transparent ML", which extends the TML calculus of Perera et al. [2012] with ML-like references and exceptions. These features potentially complicate an operational semantics, since any subexpression might modify the state or raise an exception. To avoid a proliferation of rules and threaded arguments, and to help illuminate the underlying ideas, we present the language using a variant of fine-grained call-by-value [Levy et al. 2003], which distinguishes between (pure) *expressions* and (effectful) *computations*. The syntax of the calculus, including runtime constructs, is presented in Figure 1. We omit typing rules, since static types currently play no role in the Galois slicing approach. Likewise, we omit constructs associated with isorecursive types since they contribute little in the absence of a type system. In our implementation, the source language is typed and we consider a fixed type for exceptions (for the moment, this is `string`, but any other type, such as ML's extensible exception type, would also work).

Usually we can consider a core calculus with separate expressions and computations without loss of generality because general programs can be handled by desugaring. However, since our

Expression	e	::=	$x \mid () \mid \texttt{inl}\ e \mid \texttt{inr}\ e \mid \langle e_1, e_2\rangle \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{rec}\ f(x).M$	$\mid \square$
Computation	M, N	::=	$\texttt{return}\ e \mid \texttt{let}\ x = M_1\ \texttt{in}\ M_2 \mid e_1\ e_2$	
		$\mid$	$\texttt{case}\ e\ \texttt{of}\ \{\texttt{inl}\ x.M_1; \texttt{inr}\ y.M_2\}$	
		$\mid$	$\texttt{raise}\ \ e \mid \texttt{try}\ M_1\ \texttt{with}\ x \to M_2$	
		$\mid$	$\texttt{ref}\ e \mid e_1 := e_2 \mid\ !e$	$\mid \square$
Environment	ρ, σ	::=	$\varepsilon \mid \rho[x \mapsto v]$	
Store	μ, ν	::=	$\varepsilon \mid \mu[\ell \mapsto v]$	
Set of locations	$\mathcal{L}$	::=	$\{\ell_1, \ldots, \ell_n\}$	
Value	u, v	::=	$() \mid \texttt{inl}\ v \mid \texttt{inr}\ v \mid \langle \rho, \texttt{rec}\ f(x).M\rangle \mid \ell$	$\mid \square$
Outcome	$\mathbb{k}$	::=	$\texttt{val} \mid \texttt{exn}$	
Result	R	::=	$\mathbb{k}\ v$	
Trace	T, U	::=	$\texttt{return}\ e \mid \texttt{let}_\texttt{F}(T) \mid \texttt{let}_\texttt{S}(T_1, x.T_2) \mid e_1\ e_2 \triangleright f(x).T$	
		$\mid$	$\texttt{case}_\texttt{L}(e, x.T, y) \mid \texttt{case}_\texttt{R}(e, x, y.T)$	
		$\mid$	$\texttt{raise}\ e \mid \texttt{try}_\texttt{S}(T) \mid \texttt{try}_\texttt{F}(T_1, x.T_2)$	
		$\mid$	$\texttt{ref}_\ell\ e \mid e_1 :=_\ell e_2 \mid\ !_\ell e$	$\mid \square^{\mathbb{k}}_{\mathcal{L}}$

Fig. 1. Abstract syntax

goal is to produce slices of the original program, slicing desugared programs would necessitate resugaring slices (that is, translating them back to slices of the original program, following Pombrio and Krishnamurthi [2014, 2015]). Rather than pursue this indirect approach, our implementation handles general programs directly, extrapolating from the core calculus presented in this paper. We give further details in Section 6.

The expression forms include variables, introduction forms for unit, pairs, sums, recursive functions, and side-effect-free elimination forms (pair projections). Computations represent full programs with effects: expressions are lifted to computations using the `return` operator (corresponding to monadic return), which represents the program terminating normally and returning the value of that expression, and computations are composed using $\texttt{let}\ x = M\ \texttt{in}\ N$, which corresponds to monadic bind. Abnormal termination is initiated by $\texttt{raise}\ e$, which raises e as an exception and aborts the current computation. Other computation forms include exception handling `try/with`, reference cell creation $\texttt{ref}\ e$, dereferencing $!e$, assignment $e_1 := e_2$, case analysis, and function application $e_1\ e_2$.

The evaluation of an expression yields a *value*. Values are closed and include units, pairs, injections, closures $\langle \rho, \texttt{rec}\ f(x).N\rangle$, and locations. The evaluation of a computation yields a *result*, which is either a success $\texttt{val}\ v$ or failure $\texttt{exn}\ v$, where v is a value. An *environment* ρ is a finitely supported function from variable names to values. (Recall that this means that $\rho(x)$ is defined for at most finitely many x.) We write $\rho[y \mapsto v]$ for the operation which extends ρ by mapping y to v, where $\rho(y)$ was previously undefined. A *store* is a finitely supported function from locations ℓ to values. Store update $\mu[\ell \mapsto v]$ is similar to environment update except that we do not require $\mu(\ell)$ to be undefined, as the update may overwrite the previous value of ℓ.

For present purposes, we are only concerned with slicing a computation after it has terminated, so a big-step style of operational semantics seems appropriate. The evaluation rules are given in Figure 2. (The Galois slicing approach was investigated in a small-step style for π-calculus by Perera et al. [2016].) Expression evaluation $\rho, e \Rightarrow v$ says that the expression e evaluates in environment ρ to the value v. Computation evaluation $\rho, \mu, M \Rightarrow \mu', R$, which makes use of expression evaluation, says that computation M evaluates in environment ρ and store μ to updated store μ' and result R. In the latter judgement, we choose to make explicit the derivation tree T that witnesses the evaluation

$\boxed{\rho, e \Rightarrow v}$

$$\frac{x \in \mathrm{dom}(\rho)}{\rho, x \Rightarrow \rho(x)} \qquad \frac{}{\rho, () \Rightarrow ()} \qquad \frac{}{\rho, \mathtt{rec}\ f(x).M \Rightarrow \langle \rho, \mathtt{rec}\ f(x).M \rangle} \qquad \frac{\rho, e \Rightarrow v}{\rho, \mathtt{inl}\ e \Rightarrow \mathtt{inl}\ v}$$

$$\frac{\rho, e \Rightarrow v}{\rho, \mathtt{inr}\ e \Rightarrow \mathtt{inr}\ v} \qquad \frac{\rho, e_1 \Rightarrow v_1 \qquad \rho, e_2 \Rightarrow v_2}{\rho, \langle e_1, e_2 \rangle \Rightarrow \langle v_1, v_2 \rangle} \qquad \frac{\rho, e \Rightarrow \langle v_1, v_2 \rangle}{\rho, \mathtt{fst}\ e \Rightarrow v_1} \qquad \frac{\rho, e \Rightarrow \langle v_1, v_2 \rangle}{\rho, \mathtt{snd}\ e \Rightarrow v_2}$$

$\boxed{T :: \rho, \mu, M \Rightarrow \mu', R}$

$$\frac{\rho, e \Rightarrow v}{\mathtt{return}\ e :: \rho, \mu, \mathtt{return}\ e \Rightarrow \mu, \mathtt{val}\ v}$$

$$\frac{\rho, e_1 \Rightarrow v_1 \qquad v_1 = \langle \rho', \mathtt{rec}\ f(x).M \rangle \qquad \rho, e_2 \Rightarrow v_2 \qquad T :: \rho'[f \mapsto v_1][x \mapsto v_2], \mu, M \Rightarrow \mu', R}{e_1\ e_2 \triangleright f(x).T :: \rho, \mu, e_1\ e_2 \Rightarrow \mu', R}$$

$$\frac{\rho, e \Rightarrow v}{\mathtt{raise}\ e :: \rho, \mu, \mathtt{raise}\ e \Rightarrow \mu, \mathtt{exn}\ v} \qquad \frac{\rho, e \Rightarrow v}{\mathtt{ref}_\ell\ e :: \rho, \mu, \mathtt{ref}\ e \Rightarrow \mu[\ell \mapsto v], \mathtt{val}\ \ell}\ \ell \notin \mathrm{dom}(\mu)$$

$$\frac{\rho, e_1 \Rightarrow \ell \qquad \rho, e_2 \Rightarrow v}{e_1 :=_\ell e_2 :: \rho, \mu, e_1 := e_2 \Rightarrow \mu[\ell \mapsto v], \mathtt{val}\ ()} \qquad \frac{\rho, e \Rightarrow \ell}{!_\ell e :: \rho, \mu, !e \Rightarrow \mu, \mathtt{val}\ \mu(\ell)}\ \ell \in \mathrm{dom}(\mu)$$

$$\frac{T_1 :: \rho, \mu, M_1 \Rightarrow \mu', \mathtt{val}\ v \qquad T_2 :: \rho[x \mapsto v], \mu', M_2 \Rightarrow \mu'', R}{\mathtt{let}_\mathsf{S}(T_1, x.T_2) :: \rho, \mu, \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 \Rightarrow \mu'', R}$$

$$\frac{T :: \rho, \mu, M_1 \Rightarrow \mu', \mathtt{exn}\ v}{\mathtt{let}_\mathsf{F}(T) :: \rho, \mu, \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 \Rightarrow \mu', \mathtt{exn}\ v}$$

$$\frac{T_1 :: \rho, \mu, M_1 \Rightarrow \mu', \mathtt{exn}\ v \qquad T_2 :: \rho[x \mapsto v], \mu', M_2 \Rightarrow \mu'', R}{\mathtt{try}_\mathsf{F}(T_1, x.T_2) :: \rho, \mu, \mathtt{try}\ M_1\ \mathtt{with}\ x \to M_2 \Rightarrow \mu'', R}$$

$$\frac{T_1 :: \rho, \mu, M_1 \Rightarrow \mu', \mathtt{val}\ v}{\mathtt{try}_\mathsf{S}(T_1) :: \rho, \mu, \mathtt{try}\ M_1\ \mathtt{with}\ x \to M_2 \Rightarrow \mu', \mathtt{val}\ v}$$

$$\frac{\rho, e \Rightarrow \mathtt{inl}\ v \qquad T :: \rho[x \mapsto v], \mu, M_1 \Rightarrow \mu', R}{\mathtt{case}_\mathsf{L}(e, x.T, y) :: \rho, \mu, \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inl}\ x.M_1; \mathtt{inr}\ y.M_2\} \Rightarrow \mu', R}$$

$$\frac{\rho, e \Rightarrow \mathtt{inr}\ v \qquad T :: \rho[x \mapsto v], \mu, M_2 \Rightarrow \mu', R}{\mathtt{case}_\mathsf{R}(e, x, y.T) :: \rho, \mu, \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inr}\ x.M_1; \mathtt{inr}\ y.M_2\} \Rightarrow \mu', R}$$

Fig. 2. Big-step semantics

of M (similarly to how, in type theory, typed lambda terms are essentially typing derivations); we call such a proof a *trace*. To obtain more familiar evaluation rules, it is sufficient to remove the traces from the judgments; thus we can define

$$\rho, \mu, M \Rightarrow \mu', R \iff \exists T.\ T :: \rho, \mu, M \Rightarrow \mu', R$$

The trivial computations `return` e and `raise` e evaluate the respective expressions and return them as a normal or exceptional result respectively. The evaluation of `let` $x = M$ `in` N corresponds to sequencing: the subcomputation M is evaluated first and, if it terminates successfully with `val` v, then N is evaluated next (with v substituted for x); if M terminates with a failure `exn` v, the whole `let` computation fails with `exn` v. The trace forms $\texttt{let}_\texttt{S}$ and $\texttt{let}_\texttt{F}$ correspond to these two possible evaluation outcomes.

The computation `ref` e chooses a fresh location ℓ non-deterministically, extends the store μ with a new cell at location ℓ containing the result of evaluating e, and then returns ℓ. The assignment $e_1 := e_2$ evaluates e_1 to a location ℓ and e_2 to a value v, updates the cell at ℓ with v, and returns the unit value. To evaluate a dereference $!e$, we evaluate e to a location ℓ, and then return the cell's contents $\mu(\ell)$. For convenience later, the trace forms for these three computation rules are annotated with the respective ℓ involved in the evaluation.

Function application $e_1\ e_2$ combines two pure expressions into an effectful computation as follows: first e_1 is evaluated to a closure $v_1 = \langle \rho, \texttt{rec}\ f(x).M \rangle$, where M is a computation; then e_2 is evaluated to v_2. Finally we perform the effectful evaluation of M, where recursive calls f have been replaced by the closure, and the formal argument x by the actual value v_2.

The evaluation of the exception handling `try` M_1 `with` $x \to M_2$ depends on the result of the valuation of the subcomputation M_1: if it succeeds with `val` v, we simply return this result; if it fails with `exn` v, we proceed to evaluate M_2 where v has been substituted for x. The traces $\texttt{try}_\texttt{S}$ and $\texttt{try}_\texttt{F}$ correspond to the first and second case respectively.

Finally, case analysis works as usual, taking into account that its branches are computations, whose effects are triggered after the substitution of the respective bound variables, similarly to the function application case.

Theorem 3.1. *If* $\rho, e \Rightarrow v_1$ *and* $\rho, e \Rightarrow v_2$, *then* $v_1 = v_2$.
If $T :: \rho, \mu, M \Rightarrow \mu_1, R_1$ *and* $T :: \rho, \mu, M \Rightarrow \mu_2, R_2$, *then* $\mu_1 = \mu_2$ *and* $R_1 = R_2$.

3.1 Partial expressions and partial computations

The language iTML is immediately extended by adding holes to expressions, computations, and values. This in turn induces the $\sqsubseteq$ relation, expressing the fact that two terms of the language (expressions, values, and computations) are structurally equal, save for the fact that some subterms of the right-hand side term may be matched by holes in the left-hand side term. The definition of this relation is straightforward, but verbose: we set $\square$ to be the least element and add a congruence rule per constructor. Figure 3 illustrates the cases for values, results, environments, and stores; the cases for expressions and computations are presented in an appendix in the full version of the paper.

Building on partial values, we can view environments and stores as total functions by defining $\rho(x) = \square$ and $\mu(\ell) = \square$ whenever x and ℓ are not in the domain of ρ and μ. We can then lift $\sqsubseteq$ pointwise from values to environments and stores. The least environment, mapping all variables to $\square$, is also denoted by $\square$; a similar convention applies to stores.

Meets exist for all pairs of expressions, values, computations, environments and stores. Furthermore, as we explained in Section 2, we can define the sets of prefixes of a given language

$$\boxed{v \sqsubseteq v', R \sqsubseteq R', \rho \sqsubseteq \rho', \mu \sqsubseteq \mu'}$$

$$\frac{}{\square \sqsubseteq v} \qquad \frac{}{() \sqsubseteq ()} \qquad \frac{}{\ell \sqsubseteq \ell} \qquad \frac{v_1 \sqsubseteq v_1' \qquad v_2 \sqsubseteq v_2'}{\langle v_1, v_2 \rangle \sqsubseteq \langle v_1', v_2' \rangle} \qquad \frac{v \sqsubseteq v'}{\mathtt{inl}\ v \sqsubseteq \mathtt{inl}\ v'} \qquad \frac{v \sqsubseteq v'}{\mathtt{inr}\ v \sqsubseteq \mathtt{inr}\ v'}$$

$$\frac{\rho \sqsubseteq \rho' \qquad M \sqsubseteq M'}{\langle \rho, \mathtt{rec}\ f(x).M \rangle \sqsubseteq \langle \rho', \mathtt{rec}\ f(x).M' \rangle} \qquad \frac{v \sqsubseteq v'}{\mathtt{val}\ v \sqsubseteq \mathtt{val}\ v'} \qquad \frac{v \sqsubseteq v'}{\mathtt{exn}\ v \sqsubseteq \mathtt{exn}\ v'}$$

$$\rho \sqsubseteq \rho' \iff \mathrm{dom}(\rho) = \mathrm{dom}(\rho') \wedge \forall x \in \mathrm{dom}(\rho).\ \rho(x) \sqsubseteq \rho'(x)$$

$$\mu \sqsubseteq \mu' \iff \mathrm{dom}(\mu) = \mathrm{dom}(\mu') \wedge \forall \ell \in \mathrm{dom}(\mu).\ \mu(x) \sqsubseteq \mu'(x)$$

Fig. 3. Partial value, result, environment, and store prefix relations

term:

$$\downarrow(t) = \{t' : t' \sqsubseteq t\} \qquad t = e, M, v, \rho, \mu$$

We can show that in a given prefix set, every pair of terms has a meet and a join, that is, the sets of partial terms are partonomies for the corresponding sets of ordinary terms.

Lemma 3.2. *For all expressions e, computations M, values v, environments ρ, and stores μ, the sets $\downarrow(e)$, $\downarrow(M)$, $\downarrow(v)$, $\downarrow(\rho)$ and $\downarrow(\mu)$ form complete lattices with the relation $\sqsubseteq$.*

3.2 Forward and backward slicing for expressions

Figure 4 defines the *forward slicing* and *backward slicing* relations for expressions, which are deterministic and free of side-effects and recursion. In this situation forward-slicing degenerates to a form of evaluation extended with a hole-propagation rule. The judgement $\rho, e \nearrow v$ says that partial expression e in partial environment ρ forward-slices to partial value v.

Backward-slicing for expressions is with respect to the original expression. Suppose $\rho, e \Rightarrow v$. Then for any $v' \sqsubseteq v$, the judgement $v', e \searrow \rho', e'$ says that partial value v' backward-slices along expression e to partial environment ρ' and partial expression e' with $(\rho', e') \sqsubseteq (\rho, e)$. This must be taken into account when reading the rules: for example, when we backward-slice v' with respect to an original expression x, the environment $\square[x \mapsto v']$, which maps x to v' and every other variable in the domain to $\square$, is a slice of the original ρ.

This consideration proves crucial in the backward slicing rules for pairs. To slice $\langle v_1, v_2 \rangle$ with respect to the original expression $\langle e_1, e_2 \rangle$, we first slice the two component values, obtaining ρ_1, e_1' and ρ_2, e_2', which are then recombined as $\rho_1 \sqcup \rho_2, \langle e_1', e_2' \rangle$. That is, ρ_1, e_1 tells us what part of the environment is needed to force e_1 to evaluate to v_1 and likewise for ρ_1, e_2, and we combine what we learn about ρ using $\sqcup$. As ρ_1 and ρ_2 are slices of the same original environment, the join $\rho_1 \sqcup \rho_2$ is guaranteed to exist (Lemma 3.2). The rules for slicing functions/closures are the same as given by Perera et al. [2012], and should be considered together with the rule for slicing function applications in the next section. We omit rules for primitive operations, which are handled just as in prior work [Acar et al. 2013; Perera et al. 2012].

Lemma 3.3 (Forward expression-slicing function).
(1) If $\rho, e \nearrow v$ and $\rho, e \nearrow v'$ then $v = v'$.
(2) Suppose $\rho', e' \Rightarrow v'$. If $(\rho, e) \sqsubseteq (\rho', e')$ there exists $v \sqsubseteq v'$ with $\rho, e \nearrow v$.

$$\boxed{\rho, e \nearrow v}$$

$$\frac{}{\rho, \square \nearrow \square} \qquad \frac{x \in \mathrm{dom}(\rho)}{\rho, x \nearrow \rho(x)} \qquad \frac{}{\rho, () \nearrow ()} \qquad \frac{}{\rho, \mathtt{rec}\ f(x).M \nearrow \langle \rho, \mathtt{rec}\ f(x).M \rangle} \qquad \frac{\rho, e \nearrow v}{\rho, \mathtt{inl}\ e \nearrow \mathtt{inl}\ v}$$

$$\frac{\rho, e_1 \nearrow v_1 \qquad \rho, e_2 \nearrow v_2}{\rho, \langle e_1, e_2 \rangle \nearrow \langle v_1, v_2 \rangle} \qquad \frac{\rho, e \nearrow \langle v_1, v_2 \rangle}{\rho, \mathtt{fst}\ e \nearrow v_1} \qquad \frac{\rho, e \nearrow \square}{\rho, \mathtt{fst}\ e \nearrow \square} \qquad \frac{\rho, e \nearrow \langle v_1, v_2 \rangle}{\rho, \mathtt{snd}\ e \nearrow v_2} \qquad \frac{\rho, e \nearrow \square}{\rho, \mathtt{snd}\ e \nearrow \square}$$

$$\boxed{v, e \searrow \rho, e'}$$

$$\frac{}{\square, e \searrow \square, \square} \qquad \frac{v \neq \square}{v, x \searrow \square[x \mapsto v], x} \qquad \frac{}{\langle \rho, \mathtt{rec}\ f(x).M \rangle, \mathtt{rec}\ f(x).M' \searrow \rho, \mathtt{rec}\ f(x).M}$$

$$\frac{}{(), () \searrow \square, ()} \qquad \frac{v, e \searrow \rho, e'}{\mathtt{inl}\ v, \mathtt{inl}\ e \searrow \rho, \mathtt{inl}\ e'} \qquad \frac{v, e \searrow \rho, e'}{\mathtt{inr}\ v, \mathtt{inr}\ e \searrow \rho, \mathtt{inr}\ e'}$$

$$\frac{v_1, e_1 \searrow \rho_1, e_1' \qquad v_2, e_2 \searrow \rho_2, e_2'}{\langle v_1, v_2 \rangle, \langle e_1, e_2 \rangle \searrow \rho_1 \sqcup \rho_2, \langle e_1', e_2' \rangle} \qquad \frac{\langle v, \square \rangle, e \searrow \rho, e'}{v, \mathtt{fst}\ e \searrow \rho, \mathtt{fst}\ e'} \qquad \frac{\langle \square, v \rangle, e \searrow \rho, e'}{v, \mathtt{snd}\ e \searrow \rho, \mathtt{snd}\ e'}$$

Fig. 4. Forward and backward slicing for expressions

Given Lemma 3.3, we write $\mathrm{fwd}_{\rho,e}$ for the function which take any element of $\downarrow(\rho, e)$ to its $\nearrow$-image in $\downarrow(v)$.

Lemma 3.4 (Meet-preservation). *Suppose* $\sigma, e \Rightarrow v$. *Then* $\mathrm{fwd}_{\sigma,e}$ *preserves* $\sqcap$.

Likewise, if we prioritise the use of the first rule (where $v = \square$) over others, backward-slicing determines a deterministic function, which is total if restricted to any downward-closed subset of its domain.

Lemma 3.5 (Backward expression-slicing function).
(1) If $v, e \searrow \rho, e'$ *and* $v, e \searrow \rho', e''$ *then* $(\rho, e') = (\rho', e'')$.
(2) Suppose $\rho, e \Rightarrow v$. *If* $u \sqsubseteq v$ *there exists* $(\rho', e') \sqsubseteq (\rho, e)$ *such that* $u, e \searrow \rho', e'$.

Given Lemma 3.5, we write $\mathrm{bwd}_{\rho,e}$ for the function which takes any element of $\downarrow(v)$ to its $\searrow$-image in $\downarrow(\rho, e)$. The functions $\mathrm{fwd}_{\rho,e}$ and $\mathrm{bwd}_{\rho,e}$ form a Galois connection. This is essentially a special case of the Galois connection defined by Perera et al. [2012], where the traces associated with the expression forms are simply the expressions themselves.

Theorem 3.6 (Galois connection for expression slicing). *Suppose* $\rho, e \Rightarrow v$. *Then* $\mathrm{bwd}_{\rho,e} \dashv \mathrm{fwd}_{\rho,e}$.

4 SLICING FOR REFERENCES AND EXCEPTIONS

In previous work on slicing for (pure) TML, slicing was initially focused on programs only, with forward slicing taking a program as its input, and backward slicing producing a slice of the original program. As program slices proved inadequate to explaining the behaviour of more complex code, a separate trace slicing procedure was also defined that produced a slice of the execution trace. Trace slicing has also proved useful in subsequent work on the π-calculus, where it is needed to determine a specific concurrent behaviour in an inherently nondeterministic semantics [Perera et al. 2016].

Evaluation for iTML is also non-deterministic in that it models the allocation of a new reference by chosing any unused store location. This nondeterminism is weak, in the sense that any two executions from the same initial state yield isomorphic results (up to the permutation of newly allocated locations). It is technically possible to determinise allocation and define forward slicing as a total function in the presence of references. However, if forward slicing only has access to the expression and input, it must be extremely conservative when forward slicing a hole: since we do not know the locations which were written at run-time by the missing expression, we must conservatively assume that any location may have changed, erasing the whole store. This, in turn, forces backward slicing to retain all of the write operations in the program, even those that seem to have nothing to do with the slicing criterion.

While adding exceptions to a pure language does not necessarily introduce nondeterminism, exceptions also complicate the picture for slicing considerably: if we replace a subexpression with a hole, then (in the absence of information about what happened at runtime) it is impossible to know whether that expression terminated normally or raised an exception. This means that we may be forced to retain many parts of the program solely to ensure that we can always be certain whether or not an exception was raised.

Since trace information has proven useful for implementing backward slicing even for pure programs, seems well-motivated for dealing with exceptions and references, and is in any case necessary for other features such as concurrency or true nondeterminism, we accordingly propose the following generalisation of Galois slicing, which takes explicit account of traces. Specifically, we consider *tracing* computations $C \subseteq X \times T \times Y$, where T is some set of traces that describe what happened in a given run of C, such that for any (x, t) there is a unique y such that $(x, t, y) \in C$. We assume X, Y and T are equipped with partonomies so that $\downarrow(x), \downarrow(y)$ and $\downarrow(t)$ are complete lattices for any $x \in X, y \in Y$ and $t \in T$. Given $(x, t, y) \in C$, we define a meet-preserving function $\mathsf{fwd} : \downarrow(x) \times \downarrow(t) \to \downarrow(y)$ that computes as much as possible of y given the partial information about the input in x and about the trace in t. Then a lower adjoint $\mathsf{bwd} : \downarrow(y) \to \downarrow(x) \times \downarrow(t)$ is uniquely determined, and produces the least partial input and partial trace that suffices to recompute a given partial input.

Traces, like iTML computations, are made partial by adding holes. However, rather than using a single, fully undefined trace $\Box$ that could stand for an entirely arbitrary evaluation, providing no information about its result, the parts of store that have been written, or even whether the computation succeeded or raised an exception, we provide annotated trace holes allowing for a less draconian slicing.

An annotated trace hole will be written $\Box^{\hbar}_{\mathscr{L}}$, where $\mathscr{L}$ is the set of store locations written by the otherwise unknown trace and $\hbar$ is the outcome. Unlike the unannotated hole $\Box$ used in the pure setting, annotated holes retain information about the effects and outcome of the computation. Thus, even though annotated holes do not say exactly how a computation evaluated and what its result (or exception) value was, they still disclose information about its side effects.

The $\sqsubseteq$ relation for traces is defined analogously to $\sqsubseteq$ for values, except that there is no universal least trace $\Box$. Rather, for any trace T, the hole $\Box^{\hbar}_{\mathscr{L}}$ is the least element of $\downarrow(T)$ where $\mathscr{L} = \mathsf{writes}(T)$ and $\hbar = \mathsf{outcome}(T)$:

$$\frac{\mathsf{writes}(T) = \mathscr{L} \qquad \mathsf{outcome}(T) = \hbar}{\Box^{\hbar}_{\mathscr{L}} \sqsubseteq T}$$

The auxiliary operations $\mathsf{writes}(T)$ and $\mathsf{outcome}(T)$ are defined in Figure 5. The former computes the set of locations allocated or updated by T; the latter indicates whether T returned normally or raised an exception. Thus $\Box^{\hbar}_{\mathscr{L}}$ represents the full erasure of a computation that writes to locations

$$
\begin{aligned}
\mathsf{writes}(\Box^{\hbar}_{\mathscr{L}}) &= \mathscr{L} & \mathsf{outcome}(\Box^{\hbar}_{\mathscr{L}}) &= \hbar \\
\mathsf{writes}(\mathtt{return}\ e) &= \varnothing & \mathsf{outcome}(\mathtt{return}\ e) &= \mathtt{val} \\
\mathsf{writes}(\mathtt{let}_{\mathsf{F}}(T_1)) &= \mathsf{writes}(T_1) & \mathsf{outcome}(\mathtt{let}_{\mathsf{F}}(T_1)) &= \mathtt{exn} \\
\mathsf{writes}(\mathtt{let}_{\mathsf{S}}(T_1, x.T_2)) &= \mathsf{writes}(T_1) \cup \mathsf{writes}(T_2) & \mathsf{outcome}(\mathtt{let}_{\mathsf{S}}(T_1, x.T_2)) &= \mathsf{outcome}(T_2) \\
\mathsf{writes}(e_1\ e_2 \triangleright f(x).T) &= \mathsf{writes}(T) & \mathsf{outcome}(e_1\ e_2 \triangleright f(x).T) &= \mathsf{outcome}(T) \\
\mathsf{writes}(\mathtt{case}_{\mathsf{L}}(e, x.T, y)) &= \mathsf{writes}(T) & \mathsf{outcome}(\mathtt{case}_{\mathsf{L}}(e, x.T, y)) &= \mathsf{outcome}(T) \\
\mathsf{writes}(\mathtt{case}_{\mathsf{R}}(e, x, y.T)) &= \mathsf{writes}(T) & \mathsf{outcome}(\mathtt{case}_{\mathsf{R}}(e, x, y.T)) &= \mathsf{outcome}(T) \\
\mathsf{writes}(\mathtt{raise}\ e) &= \varnothing & \mathsf{outcome}(\mathtt{raise}\ e) &= \mathtt{exn} \\
\mathsf{writes}(\mathtt{try}_{\mathsf{S}}(T_1)) &= \mathsf{writes}(T_1) & \mathsf{outcome}(\mathtt{try}_{\mathsf{S}}(T_1)) &= \mathtt{val} \\
\mathsf{writes}(\mathtt{try}_{\mathsf{F}}(T_1, x.T_2)) &= \mathsf{writes}(T_1) \cup \mathsf{writes}(T_2) & \mathsf{outcome}(\mathtt{try}_{\mathsf{F}}(T_1, x.T_2)) &= \mathsf{outcome}(T_2) \\
\mathsf{writes}(\mathtt{ref}_{\ell}\ e) &= \{\ell\} & \mathsf{outcome}(\mathtt{ref}_{\ell}\ e) &= \mathtt{val} \\
\mathsf{writes}(e_1 :=_{\ell} e_2) &= \{\ell\} & \mathsf{outcome}(e_1 :=_{\ell} e_2) &= \mathtt{val} \\
\mathsf{writes}(!_{\ell} e) &= \emptyset & \mathsf{outcome}(!_{\ell} e) &= \mathtt{val}
\end{aligned}
$$

Fig. 5. Set of store locations $\mathsf{writes}(T)$ written to by T and outcome $\mathsf{outcome}(T)$ of T.

in $\mathscr{L}$ and returns as described by $\hbar$. The $\sqsubseteq$ relation is simply the compatible closure of the above rule; the full definition is given in an appendix in the full version of the paper.

Lemma 4.1. *For all traces T, the set $\downarrow(T)$ forms a complete lattice with the relation $\sqsubseteq$.*

We will define both forward and backward slicing as judgments, and show that when $T :: \rho, \mu_1, e \Rightarrow \mu_2, R$ we can define a Galois connection bwd $\dashv$ fwd between $\downarrow(\rho, \mu_1, e, T)$ and $\downarrow(\mu_2, R)$. We first motivate our definition of forward slicing, then outline its properties, particularly meet-preservation. We then present the rules for backward slicing. Although backward slicing is uniquely determined by forward slicing, we give rules that show how to compute backward slicing more efficiently than the naive approach. The main idea, as in the pure case, is to use the trace structure to guide backward slicing. Nevertheless, due to the presence of side-effects and exceptions, there are a number of subtleties that do not arise in the pure case. The rules we give will make use of an operation for partial store erasure, which takes a store and a set of locations $\mathscr{L}$, and returns a copy of that store with all locations in $\mathscr{L}$ replaced by hole.

Definition 4.2. For partial store μ and set of locations $\mathscr{L}$, the *store erasure* operation $\mu \triangleleft \mathscr{L}$ is defined as follows:

$$\mu \triangleleft \mathscr{L} = \mu[\ell \mapsto \Box \mid \ell \in \mathscr{L}]$$

4.1 Forward slicing

In a pure language, a subexpression whose value is not needed by the rest of the computation can be sliced away safely, because we know that any other expression evaluated in its place will not raise an exception or have side-effects. However, when side-effects or exceptions are added to the picture, we need to be more careful when slicing subexpressions whose values were not needed, because the expression may have had side-effects on store locations, or the fact that the expression raised an exception may have been important to the control flow of the program. For this reason, we allow forward slicing to consult the trace, so that when information about reference side-effects or control flow is not present in the partial program, we can recover it from the trace. Thus, forward

slicing for computations is with respect to a partial trace T which determinises allocations and enables forward-slicing of store effects.

Suppose that $T :: \rho, \mu_1, M \Rightarrow \mu_2, R$. The forward slicing judgement $\rho', \mu_1', M', T' \nearrow \mu_2', R'$ takes a partial expression $e' \sqsubseteq e$, partial environment $\rho' \sqsubseteq \rho$, partial store $\mu_1' \sqsubseteq \mu_1$, and partial trace $T' \sqsubseteq T$, and should produce an updated store $\mu_2' \sqsubseteq \mu_2$ and result $R' \sqsubseteq R$.

Figures 6 and 7 define the forward slicing rules for computations, which are named for convenience. The significance of T' being partial is that forward slicing may be performed with a computation that has already been sliced, so that the forward slicing rules induce a total function from partial inputs $\downarrow(\rho, \mu_1, M, T)$ to partial outputs $\downarrow(\mu_2, R)$. The rules for forward slicing deserve explanation, since they embed important design decisions regarding how partial inputs can be used to compute partial outputs, which in turn affect the definition of backward slicing.

The first two rules (F-Trace□) and (F-Comp□) are the most important. Recall that a trace hole $\Box^{\hbar}_{\mathscr{L}}$ is annotated with the set of locations $\mathscr{L}$ written to by the trace. The (F-Trace□) rule covers the case where $T = \Box^{\hbar}_{\mathscr{L}}$. This means that we know only that while executing M, the original computation wrote to locations in $\mathscr{L}$ and eventually had outcome $\hbar$, but we have no other information about what values were written to the locations in $\mathscr{L}$ or what value was returned. Thus, we have little choice but to erase the locations in $\mathscr{L}$ in the store and yield result $\hbar\ \Box$, that is, we know that evaluation returned with outcome $\hbar$, but nothing about the value returned.

The (F-Comp□) rule covers the case where $M = \Box$. In this case, we rely on information in the trace to approximate the downstream effects on the store. We could, in principle, use T to continue recomputing, but we choose not to, since the goal of forward slicing is to show how much output can be computed from the program M. Instead, we use the auxiliary function writes to find the set of locations written to by T, and the erasure operation $\mu \triangleleft \mathscr{L}$ defined in Figure 5, to erase all locations written to by T, and return $\hbar\ \Box$ where $\hbar = \mathsf{outcome}(T)$.

Observe that the (F-Comp□) and (F-Trace□) rules overlap in the case $M = \Box, T = \Box^{\hbar}_{\mathscr{L}}$, and in this case their behaviour is identical because $\mathsf{writes}(\Box^{\hbar}_{\mathscr{L}}) = \mathscr{L}$ and $\mathsf{outcome}(\Box^{\hbar}_{\mathscr{L}}) = \hbar$.

Many of the rules are the same (modulo minor syntactic differences) as the tracing evaluation rules. The rest of the rules handle situations where a partial expression or computation forward slices to □. We discuss two of these rules, (F-Assign□) and (F-Deref□), in detail. The (F-Assign□) rule deals with the possibility that while evaluating $e_1 := e_2$, the first subexpression evaluates to □. In this case, we return □, and update the store so that $\mu(\ell) = \Box$ as well, where ℓ is the updated location recorded in the trace. This rule illustrates the benefits of using the trace: without knowing ℓ, we would be forced to conservatively set the whole store to □, since we would have no way to be sure which location was updated. Nevertheless, we do not continue evaluating using the expressions e_1', e_2' stored in the trace; they are ignored in forward slicing, but are necessary for backward slicing.

The (F-Deref□) rule deals with the possibility that when evaluating $!e$, the subexpression e evaluates to □. Again, we cannot be certain what the value of $\mu(\ell)$ is so we simply return □. Here the subscript ℓ is not needed, but again it will be needed for backward slicing.

The remaining rules (F-CaseL□), (F-CaseR□), and (F-App□) deal with the cases for case expressions or function applications in which the first argument evaluates to an unknown value or outcome □. In the case expression and function application cases, since we cannot proceed with evaluation, we proceed as in the previous case. In the other case, such as let-expressions or try-blocks, note that it is impossible for the outcome of the first subcomputation to be unknown, since we do not allow unknown outcomes □.

Forward slicing is deterministic, and total when restricted to downward-closed subsets of its domain. In particular in the $\mathtt{ref}\ \cdot$ rules, ℓ is fixed by the fact that we can consult the trace $\mathtt{ref}_{\ell}\ e'$

$\boxed{\rho, \mu, M, T \nearrow \mu', R}$

$$\textsc{F-Trace}\Box\ \frac{}{\rho, \mu, M, \Box^{\hbar}_{\mathscr{L}} \nearrow \mu \triangleleft \mathscr{L}, \hbar\ \Box} \qquad \textsc{F-Comp}\Box\ \frac{\mathscr{L} = \mathsf{writes}(T) \qquad \hbar = \mathsf{outcome}(T)}{\rho, \mu, \Box, T \nearrow \mu \triangleleft \mathscr{L}, \hbar\ \Box}$$

$$\textsc{F-Ret}\ \frac{\rho, e \nearrow v}{\rho, \mu, \mathtt{return}\ e, \mathtt{return}\ e' \nearrow \mu, \mathtt{val}\ v} \qquad \textsc{F-Let}\ \frac{\rho, \mu, M_1, T_1 \nearrow \mu', \mathtt{val}\ v \qquad \rho[x \mapsto v], \mu', M_2, T_2 \nearrow \mu'', R}{\rho, \mu, \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2, \mathtt{let}_{\mathsf{S}}(T_1, x.T_2) \nearrow \mu'', R}$$

$$\textsc{F-LetFail}\ \frac{\rho, \mu, M_1, T_1 \nearrow \mu', \mathsf{exn}\ v}{\rho, \mu, \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2, \mathtt{let}_{\mathsf{F}}(T_1) \nearrow \mu', \mathsf{exn}\ v}$$

$$\textsc{F-CaseL}\ \frac{\rho, e \nearrow \mathtt{inl}\ v \qquad \rho[x \mapsto v], \mu, M_1, T \nearrow \mu', R}{\rho, \mu, \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inl}\ x.M_1; \mathtt{inr}\ y.M_2\}, \mathtt{case}_{\mathsf{L}}(e', x.T, y) \nearrow \mu', R}$$

$$\textsc{F-CaseR}\ \frac{\rho, e \nearrow \mathtt{inr}\ v \qquad \rho[y \mapsto v], \mu, M_2, T \nearrow \mu', R}{\rho, \mu, \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inl}\ x.M_1; \mathtt{inr}\ y.M_2\}, \mathtt{case}_{\mathsf{R}}(e', x, y.T) \nearrow \mu', R}$$

$$\textsc{F-CaseL}\Box\ \frac{\rho, e \nearrow \Box \qquad \mathscr{L} = \mathsf{writes}(T) \qquad \hbar = \mathsf{outcome}(T)}{\rho, \mu, \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inl}\ x.M_1; \mathtt{inr}\ y.M_2\}, \mathtt{case}_{\mathsf{L}}(e', x.T, y) \nearrow \mu \triangleleft \mathscr{L}, \hbar\ \Box}$$

$$\textsc{F-CaseR}\Box\ \frac{\rho, e \nearrow \Box \qquad \mathscr{L} = \mathsf{writes}(T) \qquad \hbar = \mathsf{outcome}(T)}{\rho, \mu, \mathtt{case}\ e\ \mathtt{of}\ \{\mathtt{inl}\ x.M_1; \mathtt{inr}\ y.M_2\}, \mathtt{case}_{\mathsf{R}}(e', x, y.T) \nearrow \mu \triangleleft \mathscr{L}, \hbar\ \Box}$$

$$\textsc{F-App}\ \frac{\rho, e_1 \nearrow v_1 \qquad v_1 = \langle \rho', \mathtt{rec}\ f(x).M \rangle \qquad \rho, e_2 \nearrow v_2 \qquad \rho'[f \mapsto v_1][x \mapsto v_2], \mu, M, T \nearrow \mu', R}{\rho, \mu, e_1\ e_2, e_1'\ e_2' \triangleright f(x).T \nearrow \mu', R}$$

$$\textsc{F-App}\Box\ \frac{\rho, e_1 \nearrow \Box \qquad \mathscr{L} = \mathsf{writes}(T) \qquad \hbar = \mathsf{outcome}(T)}{\rho, \mu, e_1\ e_2, e_1'\ e_2' \triangleright f(x).T \nearrow \mu \triangleleft \mathscr{L}, \hbar\ \Box}$$

Fig. 6. Forward slicing for computations: holes, let-bindings, cases and function applications

of $\rho, \mu, \mathtt{ref}\ e \Rightarrow \mu'[\ell \mapsto v]$ which records the already-chosen location of ℓ. Without the trace argument, forward slicing would not be deterministic, just as ordinary evaluation is not.

Lemma 4.3 (Forward slicing function).

(1) If $\rho, \mu_1, M, T \nearrow \mu_2, R$ and $\rho, \mu_1, M, T \nearrow \mu_2', R'$ then $(\mu_2, R) = (\mu_2', R')$.

(2) Suppose $T :: \rho, \mu_1, M \Rightarrow \mu_2, R$. If $(\rho', \mu_1', M', T') \sqsubseteq (\rho, \mu_1, M, T)$ there exists $(\mu_2', R') \sqsubseteq (\mu_2, R)$ with $\rho', \mu_1', M', T' \nearrow \mu_2', R'$.

$\boxed{\rho, \mu, M, T \nearrow \mu', R}$

F-Raise
$$\frac{\rho, e \nearrow v}{\rho, \mu, \texttt{raise}\ e, \texttt{raise}\ e' \nearrow \mu, \texttt{exn}\ v}$$

F-Try
$$\frac{\rho, \mu, M_1, T_1 \nearrow \mu', \texttt{val}\ v}{\rho, \mu', \texttt{try}\ M_1\ \texttt{with}\ x \to M_2, \texttt{try}_\mathsf{S}(T_1) \nearrow \mu', \texttt{val}\ v}$$

F-TryFail
$$\frac{\rho, \mu, M_1, T_1 \nearrow \mu', \texttt{exn}\ v \qquad \rho[x \mapsto v], \mu', M_2, T_2 \nearrow \mu'', R}{\rho, \mu, \texttt{try}\ M_1\ \texttt{with}\ x \to M_2, \texttt{try}_\mathsf{F}(T_1, x.T_2) \nearrow \mu'', R}$$

F-Ref
$$\frac{\rho, e \nearrow v}{\rho, \mu, \texttt{ref}\ e, \texttt{ref}_\ell\ e' \nearrow \mu[\ell \mapsto v], \texttt{val}\ \ell}$$

F-Assign
$$\frac{\rho, e_1 \nearrow \ell \qquad \rho, e_2 \nearrow v}{\rho, \mu, e_1 := e_2, e_1' :=_\ell e_2' \nearrow \mu[\ell \mapsto v], \texttt{val}\ ()}$$

F-Assign$\square$
$$\frac{\rho, e_1 \nearrow \square}{\rho, \mu, e_1 := e_2, e_1' :=_\ell e_2' \nearrow \mu[\ell \mapsto \square], \texttt{val}\ \square}$$

F-Deref
$$\frac{\rho, e \nearrow \ell}{\rho, \mu, !e, !_\ell e' \nearrow \mu, \texttt{val}\ \mu(\ell)}$$

F-Deref$\square$
$$\frac{\rho, e \nearrow \square}{\rho, \mu, !e, !_\ell e' \nearrow \mu, \texttt{val}\ \square}$$

Fig. 7. Forward slicing for computations: exceptions and references

Given Lemma 4.3, we write fwd_T for the function which takes any element of $\downarrow(\rho', v, M', T)$ to its forward image in $\downarrow(v', R')$. (We write fwd_T instead of $\mathsf{fwd}_{\rho,\mu,M,T}$ because T provides enough information to determinise evaluation.)

Lemma 4.4 (Meet-preservation for fwd_T). *Suppose* $T :: \rho, \mu_1, M \Rightarrow \mu_2, R$ *and* $x, x' \sqsubseteq (\rho, \mu_1, M, T)$. *Then* $\mathsf{fwd}_T(x \sqcap x') = \mathsf{fwd}_T(x) \sqcap \mathsf{fwd}_T(x')$.

4.2 Backward slicing

We will define backward slicing inductively using rules for the judgment $\mu, R, T \searrow \rho, \mu', M, U$, which can be read as "To produce partial output store μ, partial result R, and partial trace T, the input environment ρ, input store μ', program M and trace U are required". The first two arguments μ and R constitute the *slicing criterion*, where μ allows us to specify what parts of the output store are of interest, and T is a trace of the computation (obtained initially from tracing evaluation of the program.)

Figure 8 defines backward slicing for computations. We explain these rules in greater detail, because there are a number of subtleties relative to the rules for slicing pure programs.

The (B-Slice$\square$) rule is applied preferentially whenever possible, to avoid a profusion of straightforward but verbose side-conditions. This rule says that if the return value of the slicing criterion is not needed *and* none of the locations $\mathscr{L}$ written to by T are needed (i.e. $\mu \triangleleft \mathscr{L} = \mu$), then we return the empty environment $\square$, unchanged store μ, empty program $\square$, and hole trace $\square^{\mathsf{k}}_{\mathscr{L}}$ recording the write set and outcome of T. The idea here is that we are allowed to slice away information that contributed only to the outcome of a computation that returns normally, as long as the result value or side effects of the computation are not needed. Thus, the annotated trace hole $\square^{\mathsf{k}}_{\mathscr{L}}$ records just enough information about T to allow us to approximate its behaviour during forward slicing.

The rule (B-Ret) for slicing $T = \texttt{return}\ e$ is straightforward; we use the expression slicing judgment. For let-binding, there are two rules: (B-Let) for $T = \texttt{let}_\mathsf{S}(T_1, x.T_2)$ when the first subexpression returns, and (B-LetFail) for $T = \texttt{let}_\mathsf{F}(T)$ when the first subexpression raises an exception. In the first case, we slice T_2 with respect to the result of the computation. This yields an environment of the form $\rho[x \mapsto v]$, where v shows what part of the value of x was required in T_2.

$$\boxed{\mu, R, T \searrow \rho, \mu', M, U}$$

$$\textsc{B-Slice}\square\ \frac{\mathscr{L} = \mathsf{writes}(T) \qquad \mu \triangleleft \mathscr{L} = \mu}{\mu, \Bbbk\ \square, T \searrow \square, \mu, \square, \square^{\Bbbk}_{\mathscr{L}}} \qquad \textsc{B-Ret}\ \frac{v, e \searrow \rho, e'}{\mu, \mathtt{val}\ v, \mathtt{return}\ e \searrow \rho, \mu, \mathtt{return}\ e', \mathtt{return}\ e'}$$

$$\textsc{B-Let}\ \frac{\mu, R, T_2 \searrow \rho_2[x \mapsto v], \mu', M_2, U_2 \qquad \mu', \mathtt{val}\ v, T_1 \searrow \rho_1, \mu'', M_1, U_1}{\mu, R, \mathtt{let_S}(T_1, x.T_2) \searrow \rho_1 \sqcup \rho_2, \mu'', \mathtt{let}\ x = M_1\ \mathtt{in}\ M_2, \mathtt{let_S}(U_1, x.U_2)}$$

$$\textsc{B-LetFail}\ \frac{\mu, \mathtt{exn}\ v, T_1 \searrow \rho, \mu', M_1, U_1}{\mu, \mathtt{exn}\ v, \mathtt{let_F}(T_1) \searrow \rho, \mu', \mathtt{let}\ x = M_1\ \mathtt{in}\ \square, \mathtt{let_F}(U_1)}$$

$$\textsc{B-CaseL}\ \frac{\mu, R, T \searrow \rho[x \mapsto v], \mu', M_1, U \qquad \mathtt{inl}\ v, e \searrow \rho', e'}{\mu, R, \mathtt{case_L}(e, x.T, y) \searrow \rho \sqcup \rho', \mu', \mathtt{case}\ e'\ \mathtt{of}\ \{\mathtt{inl}\ x.M_1; \mathtt{inr}\ y.\square\}, \mathtt{case_L}(e', x.U, y)}$$

$$\textsc{B-CaseR}\ \frac{\mu, R, T \searrow \rho[y \mapsto v], \mu', M_2, U \qquad \mathtt{inr}\ v, e \searrow \rho', e'}{\mu, R, \mathtt{case_R}(e, x, y.T) \searrow \rho \sqcup \rho', \mu', \mathtt{case}\ e'\ \mathtt{of}\ \{\mathtt{inr}\ x.\square; \mathtt{inr}\ y.M_2\}, \mathtt{case_R}(e', x, y.U)}$$

$$\textsc{B-App}\ \frac{\mu, R, T \searrow \rho[f \mapsto v_1][x \mapsto v_2], \mu', M, U \qquad v_2, e_2 \searrow \rho_2, e_2' \qquad v_1 \sqcup \langle \rho, \mathtt{rec}\ f(x).M \rangle, e_1 \searrow \rho_1, e_1'}{\mu, R, e_1\ e_2 \triangleright f(x).T \searrow \rho_1 \sqcup \rho_2, \mu', e_1'\ e_2', e_1'\ e_2' \triangleright f(x).U}$$

$$\textsc{B-Raise}\ \frac{v, e \searrow \rho, e'}{\mu, \mathtt{exn}\ v, \mathtt{raise}\ e \searrow \rho, \mu, \mathtt{raise}\ e', \mathtt{raise}\ e'}$$

$$\textsc{B-TryFail}\ \frac{\mu, R, T_2 \searrow \rho_1[x \mapsto v], \mu', M_2, U_2 \qquad \mu', \mathtt{exn}\ v, T_1 \searrow \rho_2, \mu'', M_1, U_1}{\mu, R, \mathtt{try_F}(T_1, x.T_2) \searrow \rho_1 \sqcup \rho_2, \mu'', \mathtt{try}\ M_1\ \mathtt{with}\ x \to M_2, \mathtt{try_F}(U_1, x.U_2)}$$

$$\textsc{B-Try}\ \frac{\mu, \mathtt{val}\ v, T_1 \searrow \rho, \mu', M_1, U_1}{\mu, \mathtt{val}\ v, \mathtt{try_S}(T_1) \searrow \rho, \mu', \mathtt{try}\ M_1\ \mathtt{with}\ x \to \square, \mathtt{try_S}(U_1)}$$

$$\textsc{B-Ref}\ \frac{\mu(\ell), e \searrow \rho, e'}{\mu, \mathtt{val}\ v, \mathtt{ref}_\ell\ e \searrow \rho, \mu[\ell \mapsto \square], \mathtt{ref}\ e', \mathtt{ref}_\ell\ e'}$$

$$\textsc{B-Assign}\ \frac{\mu(\ell), e_2 \searrow \rho_2, e_2' \qquad \ell, e_1 \searrow \rho_1, e_1'}{\mu, \mathtt{val}\ v, e_1 :=_\ell\ e_2 \searrow \rho_1 \sqcup \rho_2, \mu[\ell \mapsto \square], e_1' := e_2', e_1' :=_\ell\ e_2'} \qquad \textsc{B-Deref}\ \frac{\ell, e \searrow \rho, e'}{\mu, \mathtt{val}\ v, !_\ell e \searrow \rho, \mu \sqcup [\ell \mapsto v], !e', !_\ell e'}$$

Fig. 8. Backward slicing for computations

We then slice T_1 with respect to $\mathtt{val}\ v$. The partial environments ρ_1 and ρ_2 resulting from slicing the subtraces are joined, while the store μ is threaded through the slicing judgments for T_2 and T_1. The rule for $T = \mathtt{let}_{\mathsf{F}}(T')$ simply slices T' with respect to the result R (which may be $\square$ or $\mathtt{exn}\ v$).

The rules for slicing case expressions (B-CaseL), (B-CaseR) and applications (B-App) are similar to those for the corresponding constructs in pure TML; we briefly summarize them. When we slice $T = \mathtt{case}_{\mathsf{L}}(e, x.T', y)$, we slice T' with respect to the result, and obtain the value v showing what part of x is needed; we then slice e with respect to $\mathtt{inl}\ v$. The rule for $\mathtt{case}_{\mathsf{R}}(e, x, y.T')$ is symmetric. Finally, for application traces $T = e_1\, e_2 \triangleright f(x).T'$, we slice T' with respect to the outcome, and obtain from this v_1 and v_2 which show how much of the function and argument were needed for the recursive call. We also obtain ρ which shows what other values in the closure were needed and M which shows what part of the function body was needed. The argument expression e_2 is then sliced with respect to v_2 and the function expression e_1 is sliced with respect to $v_1 \sqcup \langle \rho, \mathtt{rec}\ f(x).M \rangle$. (The (B-App) rule illustrates an additional benefit of combining program and trace slicing in a single judgment; Perera et al. [2012] treated program slicing and trace slicing separately, which made it necessary to traverse the subtrace T' twice in order to perform trace slicing).

The rules for raising exceptions (B-Raise), and for dealing with `try` (B-Try), (B-TryFail), are exactly symmetric to the rules for returning normally and for let-binding.

The rules for references deserve careful examination. For reference cell creation, in rule (B-Ref) we slice the expression e with respect to the value of $\mu(\ell)$, where ℓ is the location recorded in the trace, and we update the store to map ℓ to $\square$ since ℓ is not allocated before the reference is created. The result R is irrelevant in this rule, since rerunning the reference expression will fully restore the return value ℓ.

For assignment, the rule (B-Assign) slices e_2 with respect to $\mu(\ell)$ and we slice e_1 with respect to ℓ itself. Finally, we update the store so that ℓ is mapped to $\square$; this is necessary because we have no way of knowing the value of ℓ before the assignment, and in any case it should be removed from the slicing criterion until any earlier reads from ℓ are considered. Finally, for dereferencing the rule (B-Deref) handles the case where we slice with respect to a known return value $\mathtt{val}\ v$. In this case, we can assume $v \neq \square$, since otherwise an earlier rule would apply. Thus we slice e with respect to ℓ and we add v to the slicing criterion $\mu(\ell)$.

Because of the prioritisation of the first rule, backward slicing is deterministic, and total for downward-closed subsets of its domain. Note that this preference for the first rule means that the other rules will only be used when either the value part of the result is not $\square$, or there are locations $\ell \in \mathsf{writes}(T)$ such that $\mu(\ell) \neq \square$. In particular, rules (B-Ref) and (B-Assign) will only be used when the either the returned value (ℓ or $\square$) or the value of the location ℓ that is created or assigned is part of the slicing criterion, i.e. $\mu(\ell) \neq \square$.

Lemma 4.5 (Backward slicing function).
(1) If $\mu, R, T \searrow \rho, \mu', M, U$ *and* $\mu, R, T \searrow \rho', \mu'', M', U'$ *then* $(\rho, \mu', M, U) = (\rho', \mu'', M', U')$.
(2) Suppose $T :: \rho', v, M' \Rightarrow v', R'$. *If* $(\mu, R) \sqsubseteq (v', R')$ *there exists* $(\rho, \mu', M, U) \sqsubseteq (\rho', v, M', T)$ *such that* $\mu, R, T \searrow \rho, \mu', M, U$.

Given Lemma 4.5, we write bwd_T for the function which takes any element of $\downarrow(v', R')$ to its $\searrow$-image in $\downarrow(\rho', v, M', T)$. It computes the lower adjoint of the forward slicing function for a given computation.

Theorem 4.6 (Galois connection for a computation).
Suppose $T :: \rho', v, M' \Rightarrow v', R'$.
(1) If $(\rho, \mu, M, U) \sqsubseteq (\rho', v, M', T)$ *then* $\mathsf{bwd}_T(\mathsf{fwd}_T(\rho, \mu, M, U)) \sqsubseteq (\rho, \mu, M, U)$.
(2) If $(\mu, R) \sqsubseteq (v', R')$ *then* $\mathsf{fwd}_T(\mathsf{bwd}_T(\mu, R)) \sqsupseteq (\mu, R)$.

Analogously to the Galois connection for an expression, Theorem 4.6 implies that bwd_T preserves joins (and is therefore monotonic).

Lemma 4.7 (Join-preservation for bwd_T). *Suppose* $T :: \sigma, v, M \Rightarrow v', S$ *and* $(\mu, R), (\mu', R') \sqsubseteq (v', S)$. *Then* $\mathrm{bwd}_T(\mu \sqcup \mu', R \sqcup R') = \mathrm{bwd}_T(\mu, R) \sqcup \mathrm{bwd}_T(\mu', R')$.

5 ARRAYS, SEQUENTIAL COMPOSITION, AND LOOPS

Any self-respecting imperative language includes mutable arrays, sequential composition, and loops. In this section we sketch how they can be added to our framework.

We first consider the following extension to the computations and traces to accommodate arrays:

$$\begin{array}{rcl} M & ::= & \cdots \mid \mathtt{array}(e_1, e_2) \mid e_1[e_2] \mid e_1[e_2] \leftarrow e_3 \\ v & ::= & \cdots \mid \ell\{n\} \\ T & ::= & \cdots \mid \mathtt{array}_{\ell,n}(e_1, e_2) \mid e_1[e_2]_{\ell[n]} \mid e_1[e_2] \leftarrow_{\ell[n]} e_3 \end{array}$$

where $\mathtt{array}(e_1, e_2)$ creates an array of length e_1 whose elements are initialised to e_2, while $e_1[e_2]$ gets element e_2 from array e_1 and $e_1[e_2] \leftarrow e_3$ assigns e_3 to $e_1[e_2]$. Array values $\ell\{n\}$ consist of a store location ℓ and length n. Furthermore, we extend stores to map locations to either ordinary values v or arrays $[v_1, \ldots, v_n]$. Figure 9 sketches the semantics of arrays. We omit routine additional rules for reading the length of an array. Aside from the fact that they record traces, the evaluation rules are otherwise straightforward.

Traces for array creation are annotated with the location and length of the array, while dereference and update operations are annotated with the array location and affected index. We extend the function $\mathsf{writes}(T)$ as follows:

$$\begin{array}{rcl} \mathsf{writes}(\mathtt{array}_{\ell,n}(e_1, e_2)) & = & \{\ell[0], \ldots, \ell[n-1]\} \\ \mathsf{writes}(e_1[e_2]_{\ell[i]}) & = & \{\ell[i]\} \\ \mathsf{writes}(e_1[e_2] \leftarrow_{\ell[i]} e_2) & = & \{\ell[i]\} \end{array}$$

We simply define $\mathsf{outcome}(T)$ as $\mathtt{val}$ for array traces T. (Alternatively, we could instead adjust the semantics of arrays so that exceptions are raised in the event of attempt to create an array of negative length or read or write to an out-of-bounds index. In that case we would need to annotate traces to reflect these possibilities, but we omit this added complication.)

Figure 10 shows the forward slicing rules for array constructs, which are similar to those for references. The main differences are that in the rules for dereferencing and updating, we require both the array and index parameter to be defined in order to return a value, and return $\square$ if either argument is $\square$. In that case, we also approximate the effect of the read or write on the store.

Figure 11 shows the backward slicing rules for arrays. These are again similar to those for references. In the case for array creation, we use the location and length of the created array to compute the join of all demanded parts of the initialisation expression e_2, and we also require the length n be recomputed from e_1. In the rule for backward slicing for array dereferences, we slice e_2 with respect to i and e_1 with respect to ℓ, where the annotation $\ell[i]$ records the array location and index; we also place demand v on the nth element of the array at ℓ in the store. Finally, in the backward rule for array update, using the recorded location and index $\ell[i]$, we slice e_3 with respect to the current demand on $\ell[i]$, and slice the index and array subexpressions as before. Finally we erase the ith element of the array at ℓ since its value before the update is no longer relevant until some earlier computation reads it.

Sequential composition and while-loops are definable in iTML in the usual way:

$$M_1; M_2 \quad \Longleftrightarrow \quad \mathtt{let}\ \_ = M_1\ \mathtt{in}\ M_2$$

$\boxed{T :: \rho, \mu, M \Rightarrow \mu', R}$

$$\frac{\rho, e_1 \Rightarrow n \qquad \rho, e_2 \Rightarrow v}{\mathtt{array}_{\ell,n}(e_1, e_2) :: \rho, \mu, \mathtt{array}(e_1, e_2) \Rightarrow \mu[\ell \mapsto [v, \ldots, v]], \mathtt{val}\ \ell\{n\}}$$

$$\frac{\rho, e_1 \Rightarrow \ell\{n\} \qquad \rho, e_2 \Rightarrow i \qquad 0 \le i < n}{e_1[e_2]_{\ell[i]} :: \rho, \mu, e_1[e_2] \Rightarrow \mu, \mathtt{val}\ \mu[\ell[i]]} \qquad \frac{\rho, e_1 \Rightarrow \ell\{n\} \qquad \rho, e_2 \Rightarrow i \qquad \rho, e_3 \Rightarrow v \qquad 0 \le i < n}{e_1[e_2] \leftarrow_{\ell[i]} e_3 :: \rho, \mu, e_1[e_2] \leftarrow e_3 \Rightarrow \mu[\ell[i] = v], \mathtt{val}\ ()}$$

Fig. 9. Traced evaluation for array constructs

$\boxed{\rho, \mu, M, T \nearrow \mu', R}$

$$\frac{\rho, e_1 \nearrow n \qquad \rho, e_2 \nearrow v}{\rho, \mu, \mathtt{array}(e_1, e_2), \mathtt{array}_{\ell,n}(e_1, e_2) \nearrow \mu[\ell \mapsto [v, \ldots, v]], \mathtt{val}\ \ell\{n\}}$$

$$\frac{\rho, e_1 \nearrow \square}{\rho, \mu, \mathtt{array}(e_1, e_2), \mathtt{array}_{\ell,n}(e_1, e_2) \nearrow \mu[\ell \mapsto [\square, \ldots, \square]], \mathtt{val}\ \square}$$

$$\frac{\rho, e_1 \nearrow \ell\{n\} \qquad \rho, e_2 \nearrow i}{\rho, \mu, e_1[e_2], e_1[e_2]_{\ell[i]} \nearrow \mu, \mathtt{val}\ \mu[\ell[i]]} \qquad \frac{\rho, e_1 \nearrow \square \quad \text{or} \quad \rho, e_2 \nearrow \square}{\rho, \mu, e_1[e_2], e_1[e_2]_{\ell[i]} \nearrow \mu, \mathtt{val}\ \square}$$

$$\frac{\rho, e_1 \nearrow \ell\{n\} \qquad \rho, e_2 \nearrow i \qquad \rho, e_3 \nearrow v}{\rho, \mu, e_1[e_2] \leftarrow e_3, e_1[e_2] \leftarrow_{\ell[i]} e_3 \nearrow \mu[\ell[i] \mapsto v], \mathtt{val}\ ()}$$

$$\frac{\rho, e_1 \nearrow \square \quad \text{or} \quad \rho, e_2 \nearrow \square}{\rho, \mu, e_1[e_2] \leftarrow e_3, e_1[e_2] \leftarrow_{\ell[i]} e_3 \nearrow \mu[\ell[i] \mapsto \square], \mathtt{val}\ \square}$$

Fig. 10. Forward slicing for array constructs

$\boxed{\mu, R, T \searrow \rho, \mu', M, U}$

$$\frac{\sqcup_{i=0}^{n-1} \mu(l[i]), e_2 \searrow \rho_2, e_2' \qquad n, e_1 \searrow \rho_1, e_1'}{\mu, \mathtt{val}\ v, \mathtt{array}_{\ell,n}(e_1, e_2) \searrow \rho_1 \sqcup \rho_2, \mu[\ell \mapsto \square], \mathtt{array}(e_1', e_2'), \mathtt{array}_{\ell,n}(e_1', e_2')}$$

$$\frac{i, e_2 \searrow \rho_2, e_2' \qquad \ell, e_1 \searrow \rho_1, e_1'}{\mu, \mathtt{val}\ v, e_1[e_2]_{\ell[i]} \searrow \rho_1 \sqcup \rho_2, \mu \sqcup [\ell[i] \mapsto v], e_1'[e_2'], e_1'[e_2']_{\ell[i]}}$$

$$\frac{\mu(\ell[i]), e_3 \searrow \rho_3, e_3' \qquad i, e_2 \searrow \rho_2, e_2' \qquad l, e_1 \searrow \rho_1, e_1'}{\mu, \mathtt{val}\ v, e_1[e_2] \leftarrow_{\ell[i]} e_3 \searrow \rho_1 \sqcup \rho_2 \sqcup \rho_3, \mu[\ell[i] \mapsto \square], e_1'[e_2'] \leftarrow e_3', e_1'[e_2'] \leftarrow_{\ell[i]} e_3'}$$

Fig. 11. Backward slicing for array constructs

$$\mathtt{while}\ e\ \mathtt{do}\ M \quad \Longleftrightarrow \quad (\mathtt{rec}\ \mathit{loop}(\_).\mathtt{if}\ e\ \mathtt{then}\ (M; \mathit{loop}\ ())\ \mathtt{else}\ ())\ ()$$

Our implementation supports these constructs directly, rather than via desugaring, so that slicing results in comprehensible slices in terms of these constructs. As a simple example illustrating all of

(a)

```
let x = [|0,1,2,3|] in
let i = ref 0 in
let s = ref 0 in
while (!i < 4) do (
  s := !s + x[!i];
  x[!i+1] <- s;
  i := !i + 2
)
```

(b)

```
let x = [|0,1,2,3|] in
let i = ref 0 in
let s = ref 0 in
while (!i < 4) do (
  s := !s + x[!i];
  x[!i+1] <- s;
  i := !i + 2
)
```

(c)

```
let x = [|0,1,2,3|] in
let i = ref 0 in
let s = ref 0 in
while (!i < 4) do (
  s := !s + x[!i])
  x[!i+1] <- s;
  i := !i + 2
)
```

(d)

```
let x = [|0,1,2,3|] in
let i = ref 0 in
let s = ref 0 in
while (!i < 4) do (
  s := !s + x[!i];
  x[!i+1] <- s;
  i := !i + 2
)
```

Fig. 12. Example of slicing using arrays and while-loops (a) complete program, (b) slice with respect to $!s = 2$, (c) slice with respect to $!i = 4$, (d) slice with respect to $x[3] = 2$

the above features, consider the program in Figure 12(a), which creates an array and adds up the numbers in even positions, and writes the partial sums to the odd positions. Slices are shown with respect to the final values of $!s$, $!i$, and $x[3]$ in Figure 12(b–d) respectively.

6 IMPLEMENTATION

To validate the ideas presented in the earlier sections we created an implementation[1] in Haskell (GHC 8.0.1) that allows us to run, trace, and backward slice iTML programs, along with a read-eval-print loop that allows interactive use of these features.

The calculus introduced in Section 3 is designed to reduce the number of necessary semantic rules and at the same time maintain the full expressive power of an ML-like language. The actual iTML language in our implementation does not distinguish between *expressions* and *computations*, which means that side-effecting, exception-raising computations can occur anywhere. Indeed, even constructs like nested exceptions (`raise (raise` e)) are permitted. iTML also contains integer, double, string and boolean types, arithmetic and logical operators, pair types with projections, arrays, `if` conditionals, sequencing, and loops.

To implement backward slicing algorithm we had to generalise the slicing rules from Figures 4, 8, and 11 to the full iTML language. As expected, this causes a blowup in the number of rules, from a total of twenty-five rules to over seventy cases in the actual code. Eliminating the distinction between expressions and computations also leads to the structure of traces being significantly different from the one shown in Fig. 3. In our core calculus we have two different trace forms for `let` expressions to distinguish whether a `let`-bound expression raised an exception or not, and similarly for `try-with` blocks. Given the much richer structure of expressions in the actual implementation, an approach of having several trace forms for each expression form would be impractical. So when a subexpression of a trace raises an exception we simply denote all remaining sub-traces as $\square$. So, for example, we represent $\texttt{let}_{\mathsf{F}}(T)$ as $\texttt{let}(T, x.\square)$.

To evaluate the practical usefulness of our development we decided to implement a non-trivial algorithm that relies on side-effects and may potentially raise exceptions. We picked the Gaussian elimination method for solving systems of linear equations. Our implementation is naive: it does not perform pivoting nor does it try to detect situations where a system has infinitely many solutions or no solutions at all. This means that for such systems our program will attempt a division by zero, thus raising an exception. Fig. 13(a) shows a matrix of coefficients in a 4-by-4 system of linear equations. The first iteration leads to zeroing of elements below the diagonal in the first column, but it also leads to zeroing of an element on the diagonal in the second column (Fig. 13(b), boxed).

[1] https://github.com/jstolarek/slicer

(a)

$$\left[\begin{array}{cccc|c} 3 & -1 & 2 & -1 & -13 \\ 3 & -1 & 1 & 1 & 1 \\ 1 & 2 & -1 & 2 & 21 \\ -1 & 1 & -2 & -3 & -5 \end{array}\right]$$

(b)

$$\left[\begin{array}{cccc|c} 3 & -1 & 2 & -1 & -13 \\ 0 & \boxed{0} & -1 & 2 & 14 \\ 0 & 2\frac{2}{3} & -1\frac{2}{3} & 2\frac{2}{3} & 25\frac{1}{3} \\ 0 & \frac{2}{3} & -1\frac{1}{3} & -3\frac{1}{3} & -9\frac{1}{3} \end{array}\right]$$

(c)

$$\left[\begin{array}{cccc|c} 3 & -1 & \square & \square & \square \\ 3 & -1 & \square & \square & \square \\ 1 & \boxed{2} & \square & \square & \square \\ \square & \square & \square & \square & \square \end{array}\right]$$

Fig. 13. Gaussian elimination example. (a) initial matrix; (b) after one iteration a 0 appears on the diagonal; (c) our program identifies relevant elements of original array, 2 in a box indicating place where division by zero occurred.

In the second iteration we immediately attempt to divide $2\frac{2}{3}$ by 0, which leads to an exception being raised. If we now slice the program with respect to the exception value, our implementation will identify elements of the entry matrix that were relevant in raising the exception – see Fig. 13(c), where the boxed element is where the exception is raised. Our implementation also identifies which expressions in the program were relevant to raising an exception. But since there is only one place where a division occurs in the code it is pretty obvious from the start where the division by zero must have taken place. The program slice (shown in full in an appendix in the full version of the paper) does exclude some code that was not relevant to the exception, but analysing a trace slice might be much more enlightening here.

Note that system in Fig. 13(a) has exactly one solution and if we swap the 2nd and 3rd row our implementation will find it. To test our implementation in a higher-order setting we mapped the solving function over a list of matrices that first contained a solvable modification of the system in Fig. 13(a) and then the original version that leads to division by zero. Our implementation correctly identifies the first matrix as irrelevant to the exception result and marks the same elements of second array as the ones shown in Fig. 13(c).

Our main focus has been on developing an intuitively plausible forward slicing semantics and matching backward semantics that provides useful information in the presence of side-effects, and our implementation has been helpful for establishing the usefulness of this approach. Though achieving high performance has not been our focus, it is also an important concern, so we have conducted preliminary investigations of the performance of our approach, for example by tracing and slicing computations that create lists or arrays of various lengths. Our initial approach to backward slicing recomputes writes(T) whenever the (B-Slice□) rule is attempted, and is observed to be quadratic in some cases. Understanding the performance of the Haskell implementation is nontrivial and we plan to investigate more efficient techniques in future work.

7 RELATED WORK

Galois connections are widely used in (static) program analysis in the context of *abstract interpretation* [Cousot and Cousot 1977; Darais and Van Horn 2016]. In that setting, one lattice might be the (infinite) set of sets of possible run-time behaviours of a program and another might be the (finite) set of abstractions computed by a static analysis. Abstract interpretation has also recently been related to gradual typing [Garcia et al. 2016], a technique for mixing static and dynamic type systems. Here one lattice is the set of sets of (concrete) types and another is the set of gradual types. However, both abstract interpretation and gradual typing are aimed at static analysis or typechecking of programs, whereas we consider dynamic analysis via Galois connections between lattices of partial inputs and partial outputs of a program run. On the other hand, it is an intriguing question whether the forward slicing semantics can be derived from ordinary evaluation using

Galois connections between partial objects and sets of complete objects (analogously to the AGT approach of Garcia et al. [2016] but at the expression/value level).

The application of Galois connections to program slicing for pure higher-order programs with pairs, sums and recursive datatypes was introduced by Perera et al. [2013; 2012]. Subsequent work investigated applications of related slicing techniques to security and provenance analysis [Acar et al. 2013] and explaining database queries [Cheney et al. 2014], although these papers did not employ Galois connections, opting instead for semantic notions of dependence (based on replaying traces) for which minimal slicing is undecidable. Perera et al. extended the Galois slicing approach to the π-calculus [2016]. Our work draws on their insight that trace information needs to be taken into account in the definitions of forward and backward slicing, but we consider a core language iTML for an ML-like language with imperative features, rather than the π-calculus. In principle it may be possible to translate iTML to the π-calculus, and use Perera et al.'s [2016] slicing technique on the results, but it is unclear how one might translate back to iTML, or whether the translation would introduce undesirable artefacts.

As noted earlier, there is a large literature on slicing techniques for imperative and object-oriented languages [Xu et al. 2005], but to the best of our knowledge none of this work has been extended to also handle features common to functional programming languages. Also, to the best of our knowledge the fact that optimal program slicing techniques are Galois connections has not been discussed in the slicing literature. Field and Tip [1998] present an approach to slicing for arbitrary sets of rewrite rules, in which forward slicing and backward slicing enjoy correctness and minimality properties determined by the rewriting rules. However, they considered first-order rewrite systems only, which would not suffice for a higher-order language.

The Galois slicing approach is similar in spirit to several previous papers on slicing for pure or lazy programs based on recording and analysing redex trails [Ochoa et al. 2008; Rodrigues and Barbosa 2007] and semantics-directed execution monitoring [Kishon and Hudak 1995]. Perera et al. [2012] give a more detailed comparison with this prior work. There is also a clear analogy with *declarative debugging* techniques in logic programming (including functional-logic programming languages such as Curry and Mercury). For example, tracing and dependency-tracking techniques have also been used in a tool for automated debugging in Mercury [MacLarty et al. 2005], in a system which helps programmers localise bugs by traversing an execution trace in response to programmer feedback about correct and incorrect results. Work by Silva and Chitil [2006] on combining algorithmic debugging and program slicing for pure functional programs could be generalised to automated debugging for imperative functional programs.

Biswas [1997] did consider slicing for ML programs including references and effects, but used a semantic notion of program slice for which least slices are not computable. In his approach, eliding an exception handler can allow an exception to propagate unhandled or be handled by a different handler than in the original (unsliced) program. Similarly, eliding an assignment can expose the previous value of the store location. This is in contrast with the Galois connection approach, where slicing is required to be monotone and the execution of a program slice is always a slice of the original program's execution. Thus slicing never changes the behaviour of a program, other than to elide parts in a way consistent with the original execution.

Slicing-like techniques have also been considered recently for explaining type errors. Type error explanation and diagnosis in the presence of Hindley-Milner-style type inference has been studied extensively; we mention a few closely related approaches. Haack and Wells [2004] developed methods for type error slicing for Standard ML that provide completeness and minimality guarantees; this suggests that it may be possible to view their approach as a Galois connection between lattices of programs and type errors. Seidel et al. [2016] present an approach for explaining type errors

using *dynamic witnesses*, that is, synthesised input values that illustrate how the program will go wrong. Such explanations may be more immediately useful to novices than conventional type errors, but can grow large; Seidel et al. suggest that slicing techniques may be useful for providing smaller explanations. Our work may apply here, since we can slice programs that are not well typed.

Bidirectional transformations (such as *lenses* [Foster et al. 2010]) consist of pairs of mappings between data structures that maintain some notion of consistency among them; for example, bidirectional transformations are proposed for synchronising different models of a software system, such as class diagrams and database schemas. Bidirectional transformations satisfy round-tripping laws that ensure that changes made to one side of the transformation are appropriately propagated to the other side. Among the growing literature on bidirectional programming, the work of Wang et al. [2011] seems particularly relevant, since it considers bidirectional transformations on tree-structured datatypes (e.g. abstract syntax trees). In their approach, changes to one side of the transformation can be propagated efficiently to the other side by decomposing the tree into a context (which does not change) and a focused subtree that is changed. It may make sense to view backward slicing as a special form of bidirectional transformation in which we only consider deleting subexpressions from the output; the relationship between Galois slicing and bidirectional transformations remains to be investigated.

We briefly considered the possibility of lifting slicing for the iTML core language to ML-like source programs by desugaring, slicing the desugared programs, and then somehow resugaring the sliced program back to an ML-like program. Pombrio and Krishnamurthi [2014, 2015] proposed an approach to resugaring for languages defined compositionally using hygienic macros, so that evaluation steps in the desugared language can be made meaningful in terms of the source language. Their approach establishes equational properties for round-tripping between desugaring and resugaring, similar to those encountered in bidirectional transformations or Galois connections. It would be interesting to see whether compatible desugaring/resugaring pairs can be lifted to Galois connections on partial expressions, since we could potentially then lift slicing to the source language by composing with desugaring/resugaring. Other approaches to operational semantics, such as Charguéraud's [2013] *pretty big-step semantics*, might also be worth considering.

Techniques for working with partial programs have also been considered recently by Omar et al. in the structured editor system Hazelnut [2017]. They explore usage of holes as a way to write programs in incremental steps, while guaranteeing that incomplete programs are meaningful at each intermediate editing step. Interestingly, Hazelnut allows holes to take parameters, so that a term that is not well-typed in the current context can be placed inside a parameterized hole. It may be fruitful to combine the ideas in our approach to evaluating and slicing partial programs with Hazelnut's approach to structured editing.

8 CONCLUSIONS

Despite its long history and extensive exploration in imperative or object-oriented settings, program slicing is not yet well-understood for functional languages. To date, most work on slicing for functional languages has not considered effects; the main exception is Biswas [1997], but his approach is extremely conservative in the presence of effects. On the other hand, work on slicing for imperative languages has not considered higher-order functions, datatypes or other common features of functional languages.

In this paper we generalised the Galois slicing approach, which considers fine-grained forward and backward slicing techniques as Galois connections between lattices of partial inputs and outputs, to also allow for traces that determinise and record information about the effects of computations.

We defined tracing semantics and forward and backward slicing for an imperative core language, iTML, and proved that they form a Galois connection. We have implemented and evaluated our approach on a variety of examples, providing additional confidence in its usefulness. Our main contribution is the definition of forward slicing and matching optimal backward slicing, proofs of their correctness, and experimental investigation of their qualitative usefulness.

There are a number of interesting directions for future work. Currently, there is a gap between slicing for the core language (which we use for proofs) and the source language, in which we have to handle many additional cases. It seems straightforward, albeit labour-intensive, to extend the systems and proofs; we would prefer to find a more elegant approach that allows us to lift results about slicing from the core language to the source language through resugaring and desugaring. Adapting our approach to a mainstream language may raise additional issues we have not had to consider in the core language. Extending our approach to consider other effects, objects, or concurrency appears to be a considerable challenge. Finally, we have focused on correctness and expressiveness, so finding efficient techniques for slicing that can be applied to larger programs is an important next step.

REFERENCES

Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. *Journal of Computer Security*, 21:919–969, 2013. Full version of a POST 2012 paper.

Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, January 1985. ISSN 0164-0925. doi: 10.1145/2363.2366. URL http://doi.acm.org/10.1145/2363.2366.

S. Biswas. *Dynamic Slicing in Higher-Order Programming Languages*. PhD thesis, University of Pennsylvania, 1997.

Arthur Charguéraud. Pretty-big-step semantics. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Proceedings*, pages 41–60, 2013. doi: 10.1007/978-3-642-37036-6_3. URL http://dx.doi.org/10.1007/978-3-642-37036-6_3.

James Cheney, Amal Ahmed, and Umut A. Acar. Database queries that explain their work. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 271–282, 2014.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973. URL http://doi.acm.org/10.1145/512950.512973.

David Darais and David Van Horn. Constructive Galois connections: Taming the Galois connection framework for mechanized metatheory. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 311–324, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951934. URL http://doi.acm.org/10.1145/2951913.2951934.

B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

John Field and Frank Tip. Dynamic dependence in term rewriting systems and its application to program slicing. *Information and Software Technology*, 40(11–12):609–636, November/December 1998.

Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three complementary approaches to bidirectional programming. In Jeremy Gibbons, editor, *Spring School on Generic and Indexed Programming*, volume 7470, pages 1–46. Springer, 2010. doi: 10.1007/978-3-642-32202-0_1.

Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 429–442, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837670. URL http://doi.acm.org/10.1145/2837614.2837670.

Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, March 2004. ISSN 0167-6423. doi: 10.1016/j.scico.2004.01.004. URL http://dx.doi.org/10.1016/j.scico.2004.01.004.

Amir Kishon and Paul Hudak. Semantics directed program execution monitoring. *J. Funct. Prog.*, 5(4):501–547, 1995. doi: 10.1017/S0956796800001465.

Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Inf. Comput.*, 185(2):182–210, September 2003. ISSN 0890-5401. doi: 10.1016/S0890-5401(03)00088-9. URL http://dx.doi.org/10.1016/S0890-5401(03)00088-9.

Ian MacLarty, Zoltan Somogyi, and Mark Brown. Divide-and-query and subterm dependency tracking in the mercury declarative debugger. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 59–68, New York, NY, USA, 2005. ACM. ISBN 1-59593-050-7. doi: 10.1145/1085130.1085138. URL http://doi.acm.org/10.1145/1085130.1085138.

Claudio Ochoa, Josep Silva, and Germán Vidal. Dynamic slicing of lazy functional programs based on redex trails. *Higher Order Symbol. Comput.*, 21(1-2):147–192, 2008. ISSN 1388-3690. doi: http://dx.doi.org/10.1007/s10990-008-9023-7.

Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 86–99, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009900. URL http://doi.acm.org/10.1145/3009837.3009900.

Roland Perera. *Interactive Functional Programming*. PhD thesis, University of Birmingham, Birmingham, UK, July 2013. http://etheses.bham.ac.uk/4209/.

Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *ICFP*, pages 365–376, New York, NY, USA, 2012. ACM.

Roly Perera, Deepak Garg, and James Cheney. Causally consistent dynamic slicing. In *27th International Conference on Concurrency Theory (CONCUR 2016)*, pages 18:1–18:15, 2016.

Justin Pombrio and Shriram Krishnamurthi. Resugaring: Lifting evaluation sequences through syntactic sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 361–371, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594319. URL http://doi.acm.org/10.1145/2594291.2594319.

Justin Pombrio and Shriram Krishnamurthi. Hygienic resugaring of compositional desugaring. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 75–87, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784755. URL http://doi.acm.org/10.1145/2784731.2784755.

Nuno F. Rodrigues and Luís S. Barbosa. Higher-order lazy functional slicing. *Journal of Universal Computer Science*, 13(6): 854–873, June 2007.

Eric L. Seidel, Ranjit Jhala, and Westley Weimer. Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 228–242, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951915. URL http://doi.acm.org/10.1145/2951913.2951915.

Josep Silva and Olaf Chitil. Combining algorithmic debugging and program slicing. In *PPDP*, pages 157–166, 2006.

Meng Wang, Jeremy Gibbons, and Nicolas Wu. Incremental updates for efficient bidirectional transformations. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 392–403, 2011. doi: 10.1145/2034773.2034825. URL http://doi.acm.org/10.1145/2034773.2034825.

Mark Weiser. Program slicing. In *ICSE*, pages 439–449, 1981.

Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30:1–36, March 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1050849.1050865. URL http://doi.acm.org/10.1145/1050849.1050865.